UniVerse

BASIC Reference

Part No. 70-9002A-931

NOTICE

VMARK SOFTWARE, INC., makes no warranty of any kind with regard to the material contained in this manual, including but not limited to the implied warranties of merchantability and fitness for a particular purpose.

The information contained in this manual is subject to change without notice.

This manual contains proprietary information that is protected by copyright. All rights are reserved. It may not be photocopied, reproduced, or translated, in whole or in part, without the prior express written consent of VMARK SOFTWARE, INC.

Copyright © 1988–1996 VMARK SOFTWARE, INC. All rights reserved.

Trademarks

UniVerse, UniVerse Call Interface, UniVerse Objects, UV/Net, and VMARK are trademarks of VMARK SOFTWARE, INC.

Microsoft is a registered trademark of Microsoft Corporation. Open Database Connectivity, Windows, and Windows NT are trademarks of Microsoft Corporation.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company, Ltd.

All other company or product names mentioned are trademarks or registered trademarks of their respective trademark holders.

Printing History

First Edition (70-8002A-833) for Release 8.3.3, November 1995
Second Edition (70-9002A-931) for Release 9.3.1, November 1996

How to Order Technical Documents

To order copies of documents or to obtain a catalog and price list, contact your local VMARK subsidiary or distributor, or call our main office at (508) 366-3888.

Customer Comments

We welcome your input. Please comment on this manual using the customer comment form provided in the back of the manual.

This manual is printed on recycled paper.

UniVerse BASIC Reference
Table of Contents

Preface

Organization of This Manual v
Documentation Conventions v
UniVerse Documentation vii
Related Documentation ix

Statements and Functions A–L

! statement 2
#INCLUDE statement 3
$CHAIN statement 4
$COPYRIGHT statement 5
$DEFINE statement 6
$EJECT statement 8
$ELSE statement 9
$ENDIF statement 10
$IFDEF statement 11
$IFNDEF statement 12
$INCLUDE statement 13
$INSERT statement 14
$OPTIONS statement 16
$PAGE statement 25
$UNDEFINE statement 26
* statement 27
< > operator 28
@ function 29
[ ] operator 49
ABORT statement 51
ABS function 52
ABSS function 53

ACOS function 54
ADDS function 55
ALPHA function 56
ANDS function 57
ASCII function 58
ASIN function 59
ASSIGNED function 60
assignment statements 61
ATAN function 63
AUTHORIZATION statement 64
BEGIN CASE statement 66
BEGIN TRANSACTION statement 67
BITAND function 68
BITNOT function 69
BITOR function 70
BITRESET function 71
BITSET function 72
BITTEST function 73
BITXOR function 74
BREAK statement 75
BSCAN statement 77
CALL statement 80
CASE statement 83
CATS function 86
CHAIN statement 87
CHANGE function 88
CHAR function 89
CHARS function 90
CHECKSUM function 91
CLEAR statement 92
CLEARDATA statement 93
CLEARFILE statement 94
CLEARPROMPTS statement 96

CLEARSELECT statement 97
CLOSE statement 99
CLOSESEQ statement 101
COL1 function 103
COL2 function 104
COMMIT statement 106
COMMON statement 108
COMPARE function 109
CONTINUE statement 110
CONVERT function 111
CONVERT statement 112
COS function 113
COSH function 114
COUNT function 115
COUNTS function 117
CREATE statement 119
CRT statement 120
DATA statement 121
DATE function 123
DCOUNT function 124
DEBUG statement 125
DEFFUN statement 127
DEL statement 129
DELETE function 131
DELETE statements 133
DELETELIST statement 136
DELETEU statement 137
DIMENSION statement 138
DISPLAY statement 141
DIV function 142
DIVS function 143
DOWNCASE function 144
DQUOTE function 145

DTX function 146
EBCDIC function 147
ECHO statement 148
END statement 149
END CASE statement 150
END TRANSACTION statement 151
ENTER statement 152
EOF(ARG.) function 153
EQS function 154
EQUATE statement 155
EREPLACE function 157
ERRMSG statement 158
EXCHANGE function 160
EXECUTE statement 161
EXIT statement 164
EXP function 165
EXTRACT function 166
FADD function 168
FDIV function 169
FFIX function 170
FFLT function 171
FIELD function 172
FIELDS function 174
FIELDSTORE function 175
FILEINFO function 177
FILELOCK statement 182
FILEUNLOCK statement 185
FIND statement 187
FINDSTR statement 188
FIX function 189
FLUSH statement 190
FMT function 191
FMTS function 196

FMUL function 197
FOLD function 198
FOOTING statement 199
FOR statement 203
FORMLIST statement 207
FSUB function 208
FUNCTION statement 209
GES function 211
GET statements 212
GETX statement 217
GET(ARG.) statement 218
GETLIST statement 220
GETREM function 221
GOSUB statement 222
GOTO statement 224
GROUP function 225
GROUPSTORE statement 227
GTS function 229
HEADING statement 230
HUSH statement 235
ICHECK function 237
ICONV function 239
ICONVS function 241
IF statement 243
IFS function 245
ILPROMPT function 246
INCLUDE statement 249
INDEX function 250
INDEXS function 252
INDICES function 253
INMAT function 256
INPUT statement 258
INPUTCLEAR statement 262

INPUTDISP statement 263
INPUTERR statement 264
INPUTIF statement 265
INPUTNULL statement 266
INPUTTRAP statement 267
INS statement 268
INSERT function 271
INT function 274
ISNULL function 275
ISNULLS function 276
ITYPE function 277
KEYEDIT statement 279
KEYEXIT statement 281
KEYIN function 282
KEYTRAP statement 283
LEFT function 285
LEN function 286
LENS function 287
LES function 288
LET statement 289
LN function 290
LOCATE statement 291
LOCK statement 296
LOOP statement 298
LOWER function 301
LTS function 303

Statements and Functions M–X

MAT statement 306
MATBUILD statement 308
MATCH operator 309
MATCHFIELD function 311
MATPARSE statement 313
MATREAD statements 316

MATREADL statement 320
MATREADU statement 321
MATWRITE statements 322
MATWRITEU statement 325
MAXIMUM function 326
MINIMUM function 327
MOD function 328
MODS function 329
MULS function 330
NAP statement 331
NEG function 332
NEGS function 333
NES function 334
NEXT statement 335
NOBUF statement 336
NOT function 337
NOTS function 338
NULL statement 339
NUM function 340
NUMS function 341
OCONV function 342
OCONVS function 345
ON statement 346
OPEN statement 349
OPENCHECK statement 353
OPENDEV statement 355
OPENPATH statement 357
OPENSEQ statement 360
ORS function 365
PAGE statement 366
PERFORM statement 367
PRECISION statement 369
PRINT statement 370

PRINTER statement 372
PRINTERR statement 374
PROCREAD statement 376
PROCWRITE statement 377
PROGRAM statement 378
PROMPT statement 379
PWR function 380
QUOTE function 381
RAISE function 382
RANDOMIZE statement 384
READ statements 385
READBLK statement 390
READL statement 392
READLIST statement 393
READNEXT statement 395
READSEQ statement 397
READT statement 399
READU statement 401
READV statement 402
READVL statement 403
READVU statement 404
REAL function 405
RECORDLOCKED function 406
RECORDLOCK statements 408
RELEASE statement 411
REM function 413
REM statement 414
REMOVE function 415
REMOVE statement 417
REPEAT statement 419
REPLACE function 420
RETURN statement 424
RETURN (value) statement 426

REUSE function 427
REVREMOVE statement 429
REWIND statement 431
RIGHT function 432
RND function 433
ROLLBACK statement 434
RPC.CALL function 436
RPC.CONNECT function 438
RPC.DISCONNECT function 440
SADD function 441
SCMP function 442
SDIV function 443
SEEK statement 444
SEEK(ARG.) statement 446
SELECT statements 448
SELECTE statement 451
SELECTINDEX statement 452
SELECTINFO function 453
SEND statement 454
SENTENCE function 455
SEQ function 456
SEQS function 457
SET TRANSACTION ISOLATION LEVEL statement 458
SETREM statement 460
SIN function 461
SINH function 462
SLEEP statement 463
SMUL function 464
SOUNDEX function 465
SPACE function 466
SPACES function 467
SPLICE function 468
SQRT function 469

SQUOTE function 470
SSELECT statements 471
SSUB function 474
STATUS function 475
STATUS statement 480
STOP statement 484
STORAGE statement 486
STR function 487
STRS function 488
SUBR function 489
SUBROUTINE statement 491
SUBS function 492
SUBSTRINGS function 493
SUM function 494
SUMMATION function 496
SYSTEM function 497
TABSTOP statement 501
TAN function 502
TANH function 503
TERMINFO function 504
TIME function 523
TIMEDATE function 524
TIMEOUT statement 525
TPARM function 527
TPRINT statement 530
TRANS function 532
transaction statements 534
TRANSACTION ABORT statement 535
TRANSACTION COMMIT statement 537
TRANSACTION START statement 538
TRIM function 539
TRIMB function 540
TRIMBS function 541

TRIMF function 542
TRIMFS function 543
TRIMS function 544
TTYCTL statement 545
TTYGET statement 547
TTYSET statement 553
UNASSIGNED function 555
UNLOCK statement 556
UPCASE function 557
WEOF statement 558
WEOFSEQ statement 559
WRITE statements 561
WRITEBLK statement 566
WRITELIST statement 568
WRITESEQ statement 569
WRITESEQF statement 571
WRITET statement 573
WRITEU statement 575
WRITEV statement 576
WRITEVU statement 577
XLATE function 578
XTD function 580

Index

Preface

This manual is a comprehensive reference for UniVerse BASIC. It is intended for experienced programmers, and includes explanations of all BASIC statements and functions supported by UniVerse and descriptive information regarding the use of BASIC with UniVerse in the UNIX environment.

If you have never used BASIC, read Chapters 1 and 2 in *UniVerse BASIC Programmer's Guide* before you begin. Before using any statement or function, thoroughly read its description.

Organization of This Manual

This manual contains statements and functions in alphabetical order, each beginning on a new page. At the top of each page is the syntax for the statement or function, followed by a detailed description of its use, often including references to other statements or functions that can be used with it or are helpful to know about. Examples illustrate the application of the statement or function in a program. This reference comprises two volumes:

- Volume 1 contains statements and functions beginning with symbols (such as !, $, and *) followed by statements and functions whose names begin with the letters A through L.
- Volume 2 contains statements and functions whose names begin with the letters M through X.

Documentation Conventions

This manual uses the following conventions:

Convention	Usage
Bold	In syntax, bold indicates commands, function names, keywords, and options that must be input exactly as shown. In text, bold indicates keys to press, function names, and menu selections.
UPPERCASE	In syntax, uppercase indicates UniVerse commands, keywords, and options; BASIC statements and functions; SQL statements and keywords; and ProVerb commands. In text, uppercase also indicates UniVerse identifiers such as filenames, account names, schema names, and record IDs.

Convention	Usage
Italic	In syntax, italic indicates information that you supply. In text, italic also indicates operating system commands and options, filenames, and pathnames.
`Courier`	Courier indicates examples of source code and system output.
`Courier Bold`	In examples, courier bold indicates characters that the user types or keys the user presses (for example, **`<Return>`**).
[ ]	Brackets enclose optional items. Do not type the brackets unless indicated.
{ }	Braces enclose nonoptional items from which you must select at least one. Do not type the braces.
itemA | itemB	A vertical bar separating items indicates that you can choose only one item. Do not type the vertical bar.
...	Three periods indicate that more of the same type of item can optionally follow.
➤	A right arrow between menu options indicates you should choose each option in sequence. For example, "Choose **File ➤ Exit**" means you should choose **File** from the menu bar, then choose **Exit** from the File pull-down menu.
`I`	Item mark. For example, the item mark (**`I`**) in the following string delimits elements 1 and 2, and elements 3 and 4: `1I2F3I4v5`
`F`	Field mark. For example, the field mark (**`F`**) in the following string delimits elements FLD1 and VAL1: `FLD1FVAL1vSUBV1sSUBV2`
`v`	Value mark. For example, the value mark (**`v`**) in the following string delimits elements VAL1 and SUBV1: `FLD1FVAL1vSUBV1sSUBV2`
`s`	Subvalue mark. For example, the subvalue mark (**`s`**) in the following string delimits elements SUBV1 and SUBV2: `FLD1FVAL1vSUBV1sSUBV2`
`T`	Text mark. For example, the text mark (**`T`**) in the following string delimits elements 4 and 5: `1F2s3v4T5`

The following conventions are also used:

- Syntax definitions and examples are indented for ease in reading.
- All punctuation marks included in the syntax—for example, commas, parentheses, or quotation marks—are required unless otherwise indicated.
- Syntax lines that do not fit on one line in this manual are continued on subsequent lines. The continuation lines are indented. When entering syntax, type the entire syntax entry, including the continuation lines, on the same input line.

UniVerse Documentation

UniVerse documentation includes the following:

UniVerse BASIC Programmer's Guide: This guide contains fundamental information about the UniVerse BASIC language, including data types, variables, operators, debugger commands, and conversion codes. This book is for experienced programmers.

UniVerse BASIC Reference: This two-volume manual contains reference pages for all UniVerse BASIC statements and functions, with complete syntax details. This book is for experienced programmers.

UniVerse BASIC SQL Client Interface Guide: This guide describes how to use the BASIC SQL Client Interface, which provides a programming and a user interface for UniVerse clients to access local or remote UniVerse database servers, and non-UniVerse database servers such as ORACLE, SYBASE, and INFORMIX. This book is for experienced SQL programmers.

Administering UniVerse from UNIX: This manual covers tasks performed by UniVerse Administrators using UNIX shell commands, the UniVerse System Administration menus on UNIX systems, or both. It describes such tasks as system startup and shutdown, system configuration and maintenance, system security, and peripheral maintenance.

Administering UniVerse from Windows: This manual covers tasks performed by UniVerse Administrators using the UniVerse Admin client program on a Windows system. It describes such tasks as maintaining and transferring UniVerse accounts, maintaining peripherals, backing up and restoring files, and managing file and record locks, network services, and transaction logging.

UniVerse Transaction Logging and Recovery: This manual describes the UniVerse transaction logging subsystem, including both transaction and warmstart logging and recovery. This book is for system administrators.

UniVerse System Description: This manual provides detailed and advanced information about UniVerse features and capabilities for experienced users. It describes how to use UniVerse commands, work in a UniVerse environment, create a UniVerse database, and maintain UniVerse files.

UniVerse User Reference: This manual contains reference pages for all UniVerse commands, keywords, and user records, allowing experienced users to refer to syntax details quickly.

Guide to RetrieVe: This guide describes RetrieVe, the UniVerse query language that lets users select, sort, process, and display data in UniVerse files. This book is for users who are familiar with UniVerse.

Guide to ProVerb: This guide is for application developers who want to use the ProVerb processor to create prestored procedures called procs. It describes tasks such as relational data testing, arithmetic processing, and transfers to subroutines. It also includes reference pages for all ProVerb commands.

Guide to the UniVerse Editor: This guide describes in detail how to use the Editor, allowing users to modify UniVerse files or programs. It also includes reference pages for all UniVerse Editor commands.

UniVerse SQL Administration for DBAs: This manual is for database administrators (DBAs) who are familiar with UniVerse. It describes tasks typically performed by DBAs, such as maintaining database integrity and security, and creating and modifying databases.

UniVerse SQL User Guide: This guide is for application developers who are familiar with UniVerse and want to use SQL functionality in their UniVerse applications.

UniVerse SQL Reference: This manual contains reference pages for all SQL statements and keywords, allowing experienced SQL users to refer to syntax details quickly. It includes the complete UniVerse SQL grammar in Backus Naur Form (BNF).

UniVerse Master Index: This book is a comprehensive index for UniVerse documentation.

UniVerse Quick Reference: This manual is a quick reference to all UniVerse commands and keywords. It also summarizes UniVerse SQL statements and keywords, all elements of the UniVerse BASIC language, Editor commands, ProVerb commands, file types, file dictionaries, and user exits.

Related Documentation

The following documentation is also available:

UniVerse Guide for Pick Users: This guide describes UniVerse for new UniVerse users familiar with Pick-based systems.

Moving to UniVerse from PI/open: This manual describes how to prepare the PI/open environment before converting PI/open applications to run under UniVerse. It includes step-by-step procedures for converting INFO/BASIC programs, accounts, and files. This book is for experienced PI/open users and does not assume detailed knowledge of UniVerse.

UniVerse GCI Guide: This guide describes how to use the General Calling Interface (GCI) to call subroutines written in C, C++, or FORTRAN from BASIC programs. It is for experienced programmers who are familiar with UniVerse.

UniVerse ICI Guide: This guide describes how to use the Integrated Calling Interface (ICI) to access UniVerse from external programs. It is for experienced programmers who are familiar with UniVerse.

UV/Net II Guide: This guide describes UV/Net II, the UniVerse transparent database networking facility that lets users access UniVerse files on remote systems. It is for experienced UniVerse administrators.

UV/Term Guide: This guide gives step-by-step instructions for how to install the UV/Term software, how to define terminal parameters, and how to start up and use UV/Term. It also describes the emulator window and how to use it to configure a graphical user interface.

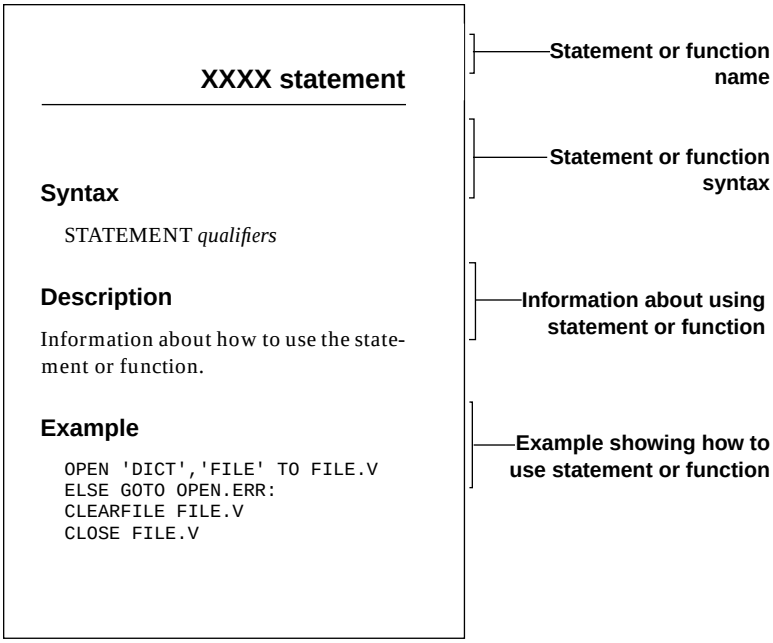
Accessing UniVerse from UNIFACE: This guide describes how to access a UniVerse database from a UNIFACE application. It is for experienced UNIFACE application developers who are new to UniVerse.

UniVerse Call Interface Guide: This guide describes how to use the UniVerse Call Interface (UCI) to write C-based client programs that use SQL function calls to access UniVerse databases. This book is for experienced SQL programmers.

Statements and Functions A–L

This volume describes the UniVerse BASIC statements and function whose names begin with symbols or the letters A through L.

Each statement and function is listed on a separate page. The sample shows a typical statement or function reference page.



! statement

Syntax

! [*comment.text*]

Description

Use the ! statement to insert a comment in a BASIC program. Comments explain or document various parts of a program. They are part of the source code only and are nonexecutable. They do not affect the size of the object code.

A comment must be a separate BASIC statement and can appear anywhere in a program. A comment must begin with one of the following comment designators:

REM * ! $*

Any text that appears between a comment designator and the end of a physical line is treated as part of the comment, not as part of the executable program. If a comment does not fit on one physical line, you can continue it on the next physical line only by starting the new line with a comment designator. If a comment appears at the end of a physical line containing an executable statement, you must put a semicolon (;) before the comment designator.

Example

The PRINT statement at the end of the third line is not executed because it follows the exclamation point on the same line and is treated as part of the comment. Lines 4, 5, and 6 show how to include a comment in the same sequence of executable statements.

```
001: PRINT "HI THERE"; ! Anything after the ! is a comment.
002: ! This line is also a comment and does not print.
003: IF 5<6 THEN PRINT "YES"; ! A comment; PRINT "PRINT ME"
004: IF 5<6 THEN
005:   PRINT "YES"; ! A comment
006:   PRINT "PRINT ME"
007: END
```

This is the program output:

```
HI THERE
YES
YES
PRINT ME
```

Syntax

#INCLUDE [*filename*] *program*

#INCLUDE *program* FROM *filename*

Description

Use the #INCLUDE statement to direct the compiler to insert the source code in the record *program* and compile it with the main program. The #INCLUDE statement differs from the $CHAIN statement in that the compiler returns to the main program and continues compiling with the statement following the #INCLUDE statement.

When *program* is specified without *filename*, *program* must be a record in the same file as the program containing the #INCLUDE statement.

If *program* is a record in a different file, the filename must be specified in the #INCLUDE statement, followed by the name of the program. The filename must specify a type 1 or type 19 file defined in the VOC file.

You can nest #INCLUDE statements.

The #INCLUDE statement is a synonym for the $INCLUDE and INCLUDE statements.

Example

```
PRINT "START"
#INCLUDE END
PRINT "FINISH"
```

When this program is compiled, the #INCLUDE statement inserts code from the program END (see the example on the END statement page). This is the program output:

```
START
THESE TWO LINES WILL PRINT ONLY
WHEN THE VALUE OF 'A' IS 'YES'.

THIS IS THE END OF THE PROGRAM
FINISH
```

$CHAIN statement

Syntax

$CHAIN [*filename*] *program*

Description

Use the $CHAIN statement to direct the compiler to read source code from *program* and compile it as if it were part of the current program. The $CHAIN statement differs from the $INCLUDE, #INCLUDE, and INCLUDE statements in that the compiler does not return to the main program. Any statements appearing after the $CHAIN statement are not compiled or executed.

When the program name is specified without a filename, the source code to insert must be in the same file as the current program.

If the source code to insert is in a different file, the $CHAIN statement must specify the name of the remote file followed by the program name. *filename* must specify a type 1 or type 19 file defined in the VOC file.

When statements in *program* generate error messages, the messages name the program containing the $CHAIN statement.

Example

```
PRINT "START"
$CHAIN END
PRINT "FINISH"
```

When this program is compiled, the $CHAIN statement inserts code from the program END (see the example on the END statement page). This is the program output:

```
START
THESE TWO LINES WILL PRINT ONLY
WHEN THE VALUE OF 'A' IS 'YES'.

THIS IS THE END OF THE PROGRAM
```

Syntax

$COPYRIGHT "*copyright.notice*"

Description

Use the $COPYRIGHT statement to specify the copyright information that is inserted in the copyright field of the object header code.

copyright.notice must be enclosed by single or double quotation marks.

The copyright field in the object code header is set to the empty string at the beginning of compilation. It remains empty until it encounters a $COPYRIGHT statement in the program.

If more than one $COPYRIGHT statement is included in the program, only the information included in the last one encountered is inserted in the object code header.

This statement is included for compatibility with existing software.

$DEFINE statement

Syntax

```
$DEFINE identifier [replacement.text]
      .
      .
      .
$IFDEF identifier
      [statements]
$ELSE
      [statements]
$ENDIF
```

Description

Use the $DEFINE statement to define identifiers that control program compilation. $UNDEFINE removes the definition of an identifier.

The $DEFINE statement has two functions:

- Defining an identifier that can then be used to control the compilation of a program
- Supplying replacement text for an identifier

identifier is the symbol to be defined. It can be any valid identifier.

replacement.text is a string of characters that the compiler uses to replace *identifier* everywhere it appears in the program containing the $DEFINE statement.

When used as a replacement text supplier, the $DEFINE statement adds the specified identifier and its associated *replacement.text* to the symbol table. Each time *identifier* is found in the program following the $DEFINE statement in which its value was set, it is replaced by *replacement.text*. If *replacement.text* is not specified, *identifier* is defined and has a null value.

One blank (or more) preceding *replacement.text* is used to separate *identifier* from *replacement.text*. Every character typed after this blank is added to *replacement.text* up to, but not including, the RETURN character, which terminates the entry of *replacement.text*.

Note: Do not use comments when supplying *replacement.text* because any comments placed after *replacement.text* are included as part of the replacement text. Any comments added to *replacement.text* may cause unexpected outcomes in your program.

Conditional Compilation

You can use $DEFINE to define an identifier that controls conditional compilation. This definition occurs irrespective of whether you supply replacement text. An identifier is defined if it has appeared in a $DEFINE statement. Once the identifier is defined, it can be used to control the conditional compilation of a program. The syntax is as follows:

$DEFINE *identifier* [*replacement.text*]

identifier is defined by the $DEFINE statement. The $IFDEF statement that begins the conditional compilation block tests *identifier* to determine whether it is defined by a $DEFINE statement. Using this syntax, *identifier* is defined so the statements between the $IFDEF and the $ELSE statements are compiled. Alternatively, if *identifier* had not been defined by a $DEFINE statement, the statements between the $ELSE and $ENDIF statements are not compiled.

Example

In this example the identifier NAME.SUFFIX is defined to have a value of PROGRAM.NAME[5]. When the compiler processes the next line, it finds the symbol NAME.SUFFIX, substitutes PROGRAM.NAME[5] in its place and continues processing with the first character of the replacement text.

```
$DEFINE NAME.SUFFIX PROGRAM.NAME[5]
IF NAME.SUFFIX = '.B' THEN
    .
    .
    .
END
    .
    .
    .
```

$EJECT statement

Syntax

$EJECT

Description

Use the $EJECT statement to begin a new page in the listing record.

This statement is a synonym for the $PAGE statement.

Syntax

$ELSE

Description

Use the $ELSE statement to introduce the alternative clauses of an $IFDEF or $IFNDEF statement.

Example

The following example determines if the identifier “modified” is defined:

```
$DEFINE modified 0
$IFDEF modified
PRINT "modified is not defined."
$ELSE
PRINT "modified is defined."
$ENDIF
```

$ENDIF statement

Syntax

$ENDIF

Description

Use the $ENDIF statement to end a conditional compilation block.

Example

The following example determines if the identifier “modified” is defined:

```
$DEFINE modified 0
$IFDEF modified
PRINT "modified is not defined."
$ELSE
PRINT "modified is defined."
$ENDIF
```

Syntax

$IFDEF *identifier*
[*statements*]

$ELSE
[*statements*]

$ENDIF

Description

Use the $IFDEF statement to test for the definition of a compile time symbol. $IFDEF tests an identifier to see if it is currently defined (that is, has appeared in a $DEFINE statement and has not been undefined).

If *identifier* is defined by a prior $DEFINE statement, all the program source lines appearing between the $IFDEF statement and the closing $ENDIF statement are compiled. If the identifier is not defined, all the lines between the $IFDEF statement and the $ENDIF statement are ignored. $IFDEF and $IFNDEF statements can be nested up to 10 deep.

Example

The following example determines if the identifier “modified” is defined:

```
$DEFINE modified 0
$IFDEF modified
PRINT "modified is defined."
$ELSE
PRINT "modified is not defined."
$ENDIF
```

$IFNDEF statement

Syntax

$IFNDEF *identifier*
[*statements*]

$ELSE
[*statements*]

$ENDIF

Description

Use the $IFNDEF statement to test for the definition of a compile time symbol. The $IFNDEF statement is a complement to the $IFDEF statement. The lines following the $IFNDEF statement are included in the compilation if the identifier is not defined. If the identifier is defined, all of the lines between the $IFNDEF statement and the $ENDIF statement are ignored. $IFDEF and $IFNDEF statements can be nested up to 10 deep.

Example

The following example determines if the identifier “modified” is not defined:

```
$DEFINE modified 0
$IFNDEF modified
PRINT "modified is not defined."
$ELSE
PRINT "modified is defined."
$ENDIF
```

Syntax

$INCLUDE [*filename*] *program*

$INCLUDE *program* FROM *filename*

Description

Use the $INCLUDE statement to direct the compiler to insert the source code in the record *program* and compile it with the main program. The $INCLUDE statement differs from the $CHAIN statement in that the compiler returns to the main program and continues compiling with the statement following the $INCLUDE statement.

When *program* is specified without *filename*, *program* must be a record in the same file as the program currently containing the $INCLUDE statement.

If *program* is a record in a different file, the filename must be specified in the $INCLUDE statement, followed by the name of the program. The filename must specify a type 1 or type 19 file defined in the VOC file.

You can nest $INCLUDE statements.

The $INCLUDE statement is a synonym for the #INCLUDE and INCLUDE statements.

Example

```
PRINT "START"
$INCLUDE END
PRINT "FINISH"
```

When this program is compiled, the $INCLUDE statement inserts code from the program END (see the example on the END statement page). This is the program output:

```
START
THESE TWO LINES WILL PRINT ONLY
WHEN THE VALUE OF 'A' IS 'YES'.

THIS IS THE END OF THE PROGRAM
FINISH
```

$INSERT statement

Syntax

$INSERT *primos.pathname*

Description

Use the $INSERT statement to direct the compiler to insert the source code contained in the file specified by *primos.pathname* and compile it with the main program. The difference between $INSERT and $INCLUDE (and its synonyms #INCLUDE and INCLUDE) is that $INSERT takes a PRIMOS pathname as an argument, whereas $INCLUDE takes a UniVerse filename and record ID. The PRIMOS pathname is converted to a pathname; any leading *> is ignored.

$INSERT is included for compatibility with Prime INFORMATION programs; the $INCLUDE statement is recommended for general use.

If *primos.pathname* is the name of the program only, it is interpreted as a relative pathname. In this case, the program must be a file in the same directory as the program containing the $INSERT statement.

You can nest $INSERT statements.

primos.pathname is converted to a valid pathname using the following conversion rules:

/ is converted to ?\

? is converted to ??

ASCII CHAR 0 (NUL) is converted to ?0

An initial . is converted to ?.

If you specify a full pathname, the > between directory names changes to a / to yield:

[*pathname*/] *program*

$INSERT uses the transformed argument directly as a pathname of the file containing the source to be inserted. It does not use the file definition in the VOC file.

Example

```
PRINT "START"
$INSERT END
PRINT "FINISH"
```

When this program is compiled, the $INSERT statement inserts code from the program END (see the example on the END statement page). This is the program output:

```
START
THESE TWO LINES WILL PRINT ONLY
WHEN THE VALUE OF 'A' IS 'YES'.

THIS IS THE END OF THE PROGRAM
FINISH
```

$OPTIONS statement

Syntax

$OPTIONS [*flavor*] [*options*]

Description

Use the $OPTIONS statement to set compile-time emulation of any UniVerse flavor. This does not allow object code compiled in one flavor to execute in another flavor. You can select individual options in a program to override the default setting.

Use the following keywords to specify *flavor*:

Keyword	Flavor
PICK	Generic Pick emulation
INFORMATION	Prime INFORMATION emulation
REALITY	REALITY emulation
IN2	Intertechnique emulation
DEFAULT	IDEAL UniVerse
PIOPEN	PI/open emulation

For instance, the following statement instructs the compiler to treat all BASIC syntax as if it were running in a PICK flavor account:

```
$OPTIONS PICK
```

Another way to select compile-time emulation is to specify one of the following keywords in field 6 of the VOC entry for the BASIC command:

```
INFORMATION.FORMAT
PICK.FORMAT
REALITY.FORMAT
IN2.FORMAT
PIOPEN.FORMAT
```

By default the VOC entry for the BASIC command corresponds with the account flavor specified when your UniVerse account was set up.

options are specified by the keywords listed in following table. To turn off an option, prefix it with a minus sign (–).

Options for the $OPTIONS Statement

Option Name	Option Letter	Description
CASE	*none*	Differentiates between uppercase and lowercase identifiers and keywords.
COMP.PRECISION	*none*	Rounds the integer at the current precision value in any comparison.
COUNT.OVLP	O	For INDEX and COUNT functions, the count overlaps.
END.WARN	R	Prints a warning message if there is no final END statement.
EXEC.EQ.PERF	P	Compiles EXECUTE as PERFORM.
EXTRA.DELIM	W	For INSERT and REPLACE functions, the compiler handles fields, values, and subvalues that contain the empty string differently from the way they are handled in the IDEAL flavor. In particular, if you specify a negative one (–1) parameter, INFORMATION and IN2 flavors add another delimiter, except when starting with an empty string.
FOR.INCR.BEF	F	Increments the index for FOR...NEXT loop before instead of after the bound checking.
FORMAT.OCONV	*none*	Lets output conversion codes be used as format masks (see the FMT function).
FSELECT	*none*	Makes the SELECT statement return the total number of records selected to the @SELECTED variable. Using this option can result in slower performance for the SELECT statement.
HEADER.BRK	*none*	Specifies the PIOPEN flavor for the I and P options to the HEADING and FOOTING keywords. This is the default for the PIOPEN flavor.

Options for the $OPTIONS Statement (Continued)

Option Name	Option Letter	Description
HEADER.DATE	D	Displays times and dates in headings in fixed format (that is, they do not change from page to page). Dates are displayed in 'D2–' format instead of 'D' format. Allows page number field specification by multiple invocations of 'P' in a single set of quotation marks.
HEADER.EJECT	H	HEADING statement causes initial page eject.
IN2.SUBSTR	T	Uses IN2 definitions for BASIC substring handling (*string***[***n,m***]**). If a single parameter is specified, a length of 1 is assumed. The size of the string expands or contracts according to the length of the replacement string.
INFO.ABORT	J	ABORT syntax follows Prime INFORMATION instead of PICK.
INFO.CONVERT	*none*	Specifies that the FMT, ICONV, and OCONV functions perform PI/open style conversions.
INFO.ENTER	*none*	Specifies the PIOPEN flavor of the ENTER statement.
INFO.INCLUDE	*none*	Processes any PRIMOS pathnames specified with the $INSERT statement.
INFO.LOCATE	L	LOCATE syntax follows Prime INFORMATION instead of REALITY. The Pick format of the LOCATE statement is always supported.
INFO.MARKS	*none*	Specifies that the LOWER, RAISE, and REMOVE functions use a smaller range of delimiters for PI/open compatibility.
INFO.MOD	*none*	Specifies the PIOPEN flavor for the MOD function. This is the default for the PIOPEN flavor.
INPUTAT	*none*	Specifies the PIOPEN flavor for the INPUT @ statement. This is the default for the PIOPEN flavor.

Options for the $OPTIONS Statement (Continued)

Option Name	Option Letter	Description
INPUT.ELSE	Y	Accepts an optional THEN...ELSE clause on INPUT statement.
INT.PRECISION	*none*	Rounds the integer at the current precision value in an INTEGER function.
LOCATE.R83	*none*	A LOCATE statement returns an “AR” or “DR” sequence value compatible with Pick, Prime INFORMATION, and PI/open systems.
NO.CASE	*none*	Does not differentiate between uppercase and lowercase in identifiers or keywords. This is the default for the PIOPEN flavor.
NO.RESELECT	U	For SELECT and SSELECT statements, active select list 0 remains active; another selection or sort is not performed. The next READNEXT statement uses select list 0.
ONGO.RANGE	G	If the value used in an ON...GOTO or ON...GOSUB is out of range, executes the next statement rather than the first or last branch.
PCLOSE.ALL	Z	The PRINTER CLOSE statement closes all print channels.
PERF.EQ.EXEC	C	PERFORM compiles as EXECUTE.
PIOPEN.INCLUDE	*none*	Processes any PRIMOS pathnames specified with the $INSERT and $INCLUDE statements.
PIOPEN.SELIDX	*none*	In the SELECTINDEX statement, removes multiple occurrences of the same record ID in an index with a multivalued field.
RADIANS	*none*	Calculates trigonometric operations using radians instead of degrees.

Options for the $OPTIONS Statement (Continued)

Option Name	Option Letter	Description
RAW.OUTPUT	*none*	Suppresses automatic mapping of system delimiters on output. When an application handles terminal control directly, RAW.OUTPUT turns off this automatic mapping.
READ.RETAIN	Q	If a READ, READU, READV, READVL, or READVU fails, the resulting variable retains its value. The variable is not set to an empty string.
REAL.SUBSTR	K	Uses REALITY flavor definitions for substring handling (*string***[***n*,*m***]**). If *m* or *n* is less than 0, the starting position for substring extraction is defined as the right side (the end) of the string.
RNEXT.EXPL	X	READNEXT returns an exploded select list.
SEQ.255	N	SEQ(" ") = 255 (instead of 0).
STATIC.DIM	M	Creates arrays at compile time, not at run time. The arrays are not redimensioned, and they do not have a zero element.
STOP.MSG	E	Causes STOP and ABORT to use the ERRMSG file to produce error messages instead of using the specified text.
SUPP.DATA.ECHO	I	Causes input statements to suppress echo from data.
TIME.MILLISECOND	*none*	Causes the SYSTEM (12) function to return the current system time in milliseconds, and the TIME function to return the current system time in seconds.
ULT.FORMAT	*none*	Format operations are compatible with Ult/ix. For example, FMT("","MR2") returns an empty string, not 0.00.
USE.ERRMSG	B	PRINT ERR prints error messages from ERRMSG.

Options for the $OPTIONS Statement (Continued)

Option Name	Option Letter	Description
VAR.SELECT	S	SELECT TO *variable* creates a local select variable instead of using numbered select lists, and READLIST reads a saved select list instead of an active numbered select list.
VEC.MATH	V	Uses vector arithmetic instructions for operating on multivalued data. For performance reasons the IDEAL flavor uses single-valued arithmetic.
WIDE.IF	*none*	Testing numeric values for true or false uses the wide zero test. In Release 6 of UniVerse, the WIDE.IF option is OFF by default. In Release 7, WIDE.IF is ON by default.

You can also set individual options by using special versions of some statements to override the current setting. These are listed as follows:

Override Versions for Statements

Statement	Equal to...
ABORTE	ABORT with $OPTIONS STOP.MSG
ABORTM	ABORT with $OPTIONS –STOP.MSG
HEADINGE	HEADING with $OPTIONS HEADER.EJECT
HEADINGN	HEADING with $OPTIONS –HEADER.EJECT
SELECTV	SELECT with $OPTIONS VAR.SELECT
SELECTN	SELECT with $OPTIONS –VAR.SELECT
STOPE	STOP with $OPTIONS STOP.MSG
STOPM	STOP with $OPTIONS –STOP.MSG

The default settings for each flavor are listed in the following table:

Default Settings of $OPTIONS Options

	IDEAL	PICK	INFO.	REALITY	IN2	PIOPEN
CASE			X			
COMP.PRECISION						
COUNT.OVLP		X		X	X	
END.WARN			X	X		X
EXEC.EQ.PERF			X			X
EXTRA.DELIM			X		X	X
FOR.INCR.BEF	X	X		X	X	
FORMAT.OCONV				X		
FSELECT						
HEADER.BRK						X
HEADER.DATE			X			X
HEADER.EJECT			X			X
IN2.SUBSTR			X		X	X
INFO.ABORT						X
INFO.CONVERT			X			X
INFO.ENTER						X
INFO.LOCATE						X
INFO.MARKS						
INFO.MOD						X
INPUTAT						X

Default Settings of $OPTIONS Options (Continued)

	IDEAL	PICK	INFO.	REALITY	IN2	PIOPEN
INPUT.ELSE		X	X			
INT.PRECISION						
LOCATE.R83						
NO.CASE						X
NO.RESELECT		X	X		X	X
NO.SMA.COMMON						
ONGO.RANGE		X			X	
PCLOSE.ALL		X		X	X	
PERF.EQ.EXEC				X		X
PIOPEN.INCLUDE						X
PIOPEN.SELIDX						X
RADIANS					X	
RAW.OUTPUT						
READ.RETAIN		X		X	X	
REAL.SUBSTR			X	X		X
RNEXT.EXPL			X			
SEQ.255		X		X	X	
STATIC.DIM		X		X	X	
STOP.MSG		X		X	X	
SUPP.DATA.ECHO		X		X	X	
ULT.FORMAT						

Default Settings of $OPTIONS Options (Continued)

	IDEAL	PICK	INFO.	REALITY	IN2	PIOPEN
USE.ERRMSG				X		
VAR.SELECT		X	X	X	X	**X**
VEC.MATH			X			**X**
WIDE.IF	X	X	X	X	X	

Example

```
>ED BP OPT
4 lines long.
----: P
0001: $OPTIONS INFORMATION
0002: A='12'
0003: B='14'
0004: PRINT A,B
Bottom at line 4
----: Q
>BASIC BP OPT
Compiling: Source = 'BP/OPT', Object = 'BP.O/OPT'

@EOF    WARNING: Final 'END' statement not found.

Compilation Complete.
>ED BP OPT
4 lines long.
----: P
0001: $OPTIONS PICK
0002: A='12'
0003: B='14'
0004: PRINT A,B
Bottom at line 4
----: Q
>BASIC BP OPT
Compiling: Source = 'BP/OPT', Object = 'BP.O/OPT'
Compilation Complete.
```

$PAGE statement

The $PAGE statement is a synonym for the $EJECT statement.

$UNDEFINE statement

Syntax

$UNDEFINE *identifier*

Description

Use the $UNDEFINE statement to remove the definition of identifiers set with the $DEFINE statement. The $UNDEFINE statement removes the definition of *identifier* from the symbol table if it appeared in a previous $DEFINE statement. If the identifier was not previously defined, $UNDEFINE has no effect.

identifier is the identifier whose definition is to be deleted from the symbol table.

Conditional Compilation

When you use $UNDEFINE to remove an identifier, you can specify conditional compilation. In this syntax, *identifier* is first defined by the $DEFINE statement. Subsequently the $UNDEFINE statement deletes the definition of *identifier* from the symbol table. The $IFDEF statement that begins the conditional compilation block tests identifier to determine whether it has been defined. The $IFDEF statement determines that *identifier* is not defined, so the statements between the $ELSE and $ENDIF statements are compiled. Alternatively, you can use the $IFNDEF statement to test whether *identifier* has been defined.

Syntax

* [*comment.text*]

Description

Use the * statement to insert a comment in a BASIC program. Comments explain or document various parts of a program. They are part of the source code only and are nonexecutable. They do not affect the size of the object code.

A comment must be a separate BASIC statement, and can appear anywhere in a program. A comment must begin with one of the following comment designators:

REM * ! $*

Any text that appears between a comment designator and the end of a physical line is treated as part of the comment, not as part of the executable program. If a comment does not fit on one physical line, you can continue it on the next physical line only by starting the new line with a comment designator. If a comment appears at the end of a physical line containing an executable statement, you must put a semicolon (;) before the comment designator.

Example

The PRINT statement at the end of the third line is not executed because it follows the asterisk on the same line and is treated as part of the comment. Lines 4, 5, and 6 show how to include a comment in the same sequence of executable statements.

```
001: PRINT "HI THERE"; * Anything after the * is a comment
002: * This line is also a comment and does not print.
003: IF 5<6 THEN PRINT "YES"; * A comment; PRINT "PRINT ME"
004: IF 5<6 THEN
005:   PRINT "YES"; * A comment
006:   PRINT "PRINT ME"
007: END
```

This is the program output:

```
HI THERE
YES
YES
PRINT ME
```

< > operator

Syntax

variable < *field#* [*,value#* [*,subvalue#*]] >

Description

Use the < > operator (angle brackets) to extract or replace elements of a dynamic array.

variable specifies the dynamic array containing the data to be changed.

field#, *value#*, and *subvalue#* are delimiter expressions.

Angle brackets to the left of an assignment operator change the specified data in the dynamic array according to the assignment operator. For examples, see the REPLACE function.

Angle brackets to the right of an assignment operator indicate that an EXTRACT function is to be performed. For examples, see the EXTRACT function.

Syntax

@(*column* [,*row*])

@(–*code* [,*arg*])

Description

Use the @ function with the PRINT statement to control display attributes, screen display, and cursor postioning.

Note: You can save processing time by assigning the result of a commonly used @ function, such as @(–1), to a variable, rather than reevaluating the function each time it is used.

column defines a screen column position.

row defines a screen row position.

–*code* is the terminal control code that specifies a particular screen or cursor function.

arg specifies further information for the screen or cursor function specified in –*code*.

Cursor Positioning

You position the cursor by specifying a screen column and row position using the syntax @(*column* **[**,*row***]**). If you do not specify a row, the current row is the default. The top line is row 0, the leftmost column is column 0. If you specify a column or row value that is out of range, the effect of the function is undefined.

If you use the @ function to position the cursor, automatic screen pagination is disabled.

Screen and Cursor Controls

You can use the @ function with terminal control codes to specify various cursor and display operations using the syntax @(–*code* **[**,*arg***]**).

If you want to use mnemonics rather than the code numbers, you can use an insert file of equate names by specifying either of the following options when you compile your program:

`$INCLUDE UNIVERSE.INCLUDE ATFUNCTIONS.H`

`$INCLUDE SYSCOM ATFUNCTIONS.INS.IBAS` (PIOPEN flavor only)

Note: Not all terminal control codes are supported by all terminal types. If the current terminal type does not support the code you specified, the function returns an empty string. You can use this to test whether your program operates correctly on a particular terminal, and whether you need to code any alternative actions.

If you issue multiple video attributes (such as blink and reverse video) at the same time, the result is undefined. See the description of the @(IT$VIDEO) function for details of additive attributes.

The following table summarizes the characteristics of the terminal control codes, and the sections following the table give more information on each equate name:

Terminal Control Codes

Integer	Equate Name	Function	Argument
–1	IT$CS	Screen clear and home	
–2	IT$CAH	Cursor home	
–3	IT$CLEOS	Clear to end of screen	
–4	IT$CLEOL	Clear to end of line	
–5	IT$SBLINK	Start blink	
–6	IT$EBLINK	Stop blink	
–7	IT$SPA	Start protect	
–8	IT$EPA	Stop protect	
–9	IT$CUB	Back space one character	Number of characters to back space

Terminal Control Codes (Continued)

Integer	Equate Name	Function	Argument
–10	IT$CUU	Move up one line	Number of lines to move
–11	IT$SHALF	Start half-intensity	
–12	IT$EHALF	Stop half-intensity	
–13	IT$SREV	Start reverse video	
–14	IT$EREV	Stop reverse video	
–15	IT$SUL	Start underlining	
–16	IT$EUL	Stop underlining	
–17	IT$IL	Insert line	Number of lines to insert
–18	IT$DL	Delete line	Number of lines to delete
–19	IT$ICH	Insert character	Number of lines to insert
–20	IT$SIRM	Set insert/replace mode	
–21	IT$RIRM	Reset insert/replace mode	
–22	IT$DCH	Delete character	Number of characters to delete
–23	IT$AUXON	Auxiliary port on	
–24	IT$AUXOFF	Auxiliary port off	
–25	IT$TRON	Transparent auxiliary port on	
–26	IT$TROFF	Transparent auxiliary port off	
–27	IT$AUXDLY	Auxiliary port delay time	
–28	IT$PRSCRN	Print screen	
–29	IT$E80	Enter 80-column mode	

Terminal Control Codes (Continued)

Integer	Equate Name	Function	Argument
–30	IT$E132	Enter 132-column mode	
–31	IT$RIC	Reset inhibit cursor	
–32	IT$SIC	Set inhibit cursor	
–33	IT$CUD	Cursor down	Number of lines to move cursor
–34	IT$CUF	Cursor forward	Number of places to move cursor forward
–35	IT$VIDEO	Set video attributes	Additive attribute value
–36	IT$SCOLPR	Set color pair	Predefined color pairing
–37	IT$FCOLOR	Set foreground color	Foreground color code
–38	IT$BCOLOR	Set background color	Background color code
–39	IT$SLINEGRFX	Start line graphics	
–40	IT$ELINEGRFX	End line graphics	
–41	IT$LINEGRFXCH	Line graphics character	The required graphics character
–42	IT$DMI	Disable manual input	
–43	IT$EMI	Enable manual input	
–44	IT$BSCN	Blank screen	
–45	IT$UBS	Unblank screen	
–48	IT$SU	Scroll up	Number of lines to scroll
–49	IT$SD	Scroll down	Number of lines to scroll
–50	IT$SR	Scroll right	Number of columns to scroll
–51	IT$SL	Scroll left	Number of columns to scroll

Terminal Control Codes (Continued)

Integer	Equate Name	Function	Argument
–54	IT$SLT	Set line truncate	
–55	IT$RLT	Reset line truncate	
–56	IT$SNK	Set numeric keypad	
–57	IT$RNK	Reset numeric keypad	
–58	IT$SBOLD	Start bold	
–59	IT$EBOLD	End bold	
–60	IT$SSECUR	Start secure mode	
–61	IT$ESECUR	End secure mode	
–62	IT$SSCRPROT	Start screen protect mode	
–63	IT$ESCRPROT	End screen protect mode	
-64	IT$SLD	System line display	
–65	IT$SLR	System line reset	
–66	IT$SLS	System line set	
–70	IT$CHA	Cursor horizontal absolute	Column number to position cursor
–71	IT$ECH	Erase character	Number of characters to erase
–74	IT$NPC	Character to substitute for nonprinting character	
–75	IT$DISPLAY	EDFS main display attributes	
–76	IT$MINIBUF	EDFS mini-buffer display attributes	
–77	IT$LOKL	Lock line	The line number
–78	IT$UNLL	Unlock line	The line number

Terminal Control Codes (Continued)

Integer	Equate Name	Function	Argument
–79	IT$MARKSUBS	Display marks	
–80 through –100		Reserved for VMARK	
–101 through –128	IT$USERFIRST IT$USERLAST	Available for general use	

Screen Clear and Home @(IT$CS)

Clears the screen and positions the cursor in the upper-left corner.

Cursor Home @(IT$CAH)

Moves the cursor to the upper-left corner of the screen.

Clear to End of Screen @(IT$CLEOS)

Clears the current screen line starting at the position under the cursor to the end of that line and clears all lines below that line. The cursor does not move.

Clear to End of Line @(IT$CLEOL)

Clears the current screen line starting at the position under the cursor to the end of that line. The cursor does not move.

Start Blink @(IT$SBLINK)

Causes any printable characters that are subsequently displayed to blink. If you move the cursor before issuing the stop blink function, @(IT$EBLINK), the operation of the @(IT$SBLINK) code is undefined.

Stop Blink @(IT$EBLINK)

Stops blink mode. If a start blink function, @(IT$SBLINK), was not transmitted previously, the effect of this sequence is undefined.

Start Protect @(IT$SPA)

Protects all printable characters that are subsequently displayed from update until the characters are erased by one of the clear functions @(IT$CS), @(IT$CLEOS), or @(IT$CLEOL). If you move the cursor before issuing the stop protect function, @(IT$EPA), the operation of this code is undefined. The start protect function is useful only for terminals that are in block mode.

Stop Protect @(IT$EPA)

Stops the protect mode. If a start protect string was not previously transmitted, the effect of this sequence is undefined. The stop protect function is useful only for terminals that are in block mode.

Back Space One Char @(IT$CUB)

Moves the cursor one position to the left without deleting any data. For *m* greater than 0, the function @(IT$CUB, *m*) moves the cursor *m* positions to the left. In moving to the left, the cursor cannot move beyond the start of the line.

Move Up One Line @(IT$CUU)

Moves the cursor up one line toward the top of the screen. For *m* greater than 0, the function @(IT$CUU, *m*) moves the cursor up *m* lines. The cursor remains in the same column, and cannot move beyond the top of the screen.

Start Half-intensity @(IT$SHALF)

Causes all printable characters that are subsequently displayed to be displayed at reduced intensity. If a cursor-positioning sequence is used before the stop half-intensity function, @(IT$EHALF), the operation of this function is undefined.

Stop Half-intensity @(IT$EHALF)

Terminates half-intensity mode. The effect of this sequence is unspecified if a start half-intensity string was not previously transmitted.

Start Reverse Video @(IT$SREV)

Causes printable characters that are subsequently displayed to be displayed with all pixels inverted. If a cursor-positioning sequence is used before the stop reverse video function, @(IT$EREV), the operation of this function is undefined.

Stop Reverse Video @(IT$EREV)

Terminates reverse video mode. If a start reverse video function, @(IT$SREV), was not previously transmitted, the effect of this sequence is undefined.

Start Underlining @(IT$SUL)

Causes all subsequent printable characters to be underlined when displayed. If a cursor-positioning sequence is used before the stop underlining function, @(IT$EUL), the operation of this function is undefined.

Stop Underlining @(IT$EUL)

Terminates the underlining mode established by a start underlining function, @(IT$SUL). The effect of this sequence is unspecified if a start underlining string was not previously transmitted.

Insert Line @(IT$IL)

Inserts a blank line at the current cursor position. For *m* greater than 0, the function @(IT$IL, *m*) inserts m blank lines at the current cursor position. If *m* is omitted, the default is 1. The effect when *m* is less than 1 is undefined. All lines from the current cursor position to the end of the screen scroll down. The bottom *m* lines on the screen are lost.

Delete Line @(IT$DL)

Deletes the line at the current cursor position; the function @(IT$DL, 1) has the same effect. For *m* greater than 1, the lines above the current line are deleted until *m* minus 1 lines have been deleted or the top of the file has been reached, whichever occurs first. All lines below the current cursor position scroll up. The last lines on the screen are cleared.

Insert Character @(IT$ICH)

Inserts a space at the current cursor position. All characters from the cursor position to the right edge of the screen are shifted over one character to the right. Any character at the rightmost edge of the screen is lost. For *m* greater than 0, the function @(IT$ICH, *m*) inserts m spaces at the current cursor position, shifting the other characters accordingly.

Set Insert/Replace Mode @(IT$SIRM)

Starts insert character mode. Characters sent to the terminal screen are inserted at the current cursor position instead of overwriting the character under the cursor. The characters under and to the right of the cursor are shifted over one character to the right for each character transmitted, and any character at the rightmost edge of the screen is lost.

Reset Insert/Replace Mode @(IT$RIRM)

Turns off insert character mode. Characters sent to the terminal screen overwrite the characters at the current cursor position.

Delete Character @(IT$DCH)

Deletes the character at the current cursor position. All characters to the right of the cursor move one space to the left, and the last character position on the line is made blank. For *m* greater than 1, the function @(IT$DCH, *m*) deletes further characters, to the right of the original position, until *m* characters have been deleted altogether or until the end of the display has been reached, whichever occurs first.

Auxiliary Port On @(IT$AUXON)

Enables the auxiliary (printer) port on the terminal. All characters sent to the terminal are displayed on the screen and also copied to the auxiliary port.

Auxiliary Port Off @(IT$AUXOFF)

Disables the auxiliary (printer) port on the terminal, and stops the copying of the character stream to the auxiliary port.

Transparent Auxiliary Port On @(IT$TRON)

Places the auxiliary (printer) port on the terminal in transparent mode. All characters sent to the terminal are sent only to the auxiliary port and are not displayed on the terminal screen.

Transparent Auxiliary Port Off @(IT$TROFF)

Disables the auxiliary (printer) port on the terminal and enables the display of the character stream on the terminal screen.

Auxiliary Delay Time @(IT$AUXDLY)

Sets a time, in milliseconds, that an application should pause after enabling or disabling the auxiliary port. The value of this function is an integer in the range 0 through 32,767. The function is used in conjunction with the !SLEEP$ subroutine; for example:

```
PRINT @(IT$AUXON):;CALL !SLEEP$(@(IT$AUXDLY))
```

Print Screen @(IT$PRSCRN)

Copies the contents of the screen to the auxiliary port. The function does not work for some terminals while echo delay is enabled.

Enter 80-Column Mode @(IT$E80)

Starts 80-column mode. On some terminals it can also clear the screen.

Enter 132-Column Mode @(IT$E132)

Starts 132-column mode. On some terminals it can also clear the screen.

Reset Inhibit Cursor @(IT$RIC)

Turns the cursor on.

Set Inhibit Cursor @(IT$SIC)

Turns the cursor off.

Cursor Down @(IT$CUD)

Moves the cursor down one line. For *m* greater than 0, the function @(IT$CUD, *m*) moves the cursor down *m* lines. The cursor remains in the same column, and cannot move beyond the bottom of the screen.

Cursor Forward @(IT$CUF)

Moves the cursor to the right by one character position without overwriting any data. For *m* greater than 0, the function @(IT$CUF, *m*) moves the cursor *m* positions to the right. The cursor cannot move beyond the end of the line.

Set Video Attributes @(IT$VIDEO)

Is an implementation of the ANSI X3.64-1979 and ISO 6429 standards for the video attribute portion of Select Graphic Rendition. It always carries an argument *m* that is an additive key consisting of one or more of the following video attribute keys:

Value	Name	Description
0	IT$NORMAL	Normal
1	IT$BOLD	Bold
2	IT$HALF	Half-intensity
4	IT$STANDOUT	Enhanced
4	IT$ITALIC	Italic
8	IT$ULINE	Underline
16	IT$SLOWBLINK	Slow blink
32	IT$FASTBLINK	Fast blink
64	IT$REVERSE	Reverse video
128	IT$BLANK	Concealed
256	IT$PROTECT	Protected
572	IT$ALTCHARSET	Alternative character set

For example:

```
PRINT @(IT$VIDEO,IT$HALF+IT$ULINE+IT$REVERSE)
```

In this example, *m* is set to 74 (2 + 8 + 64) for half-intensity underline display in reverse video. Note that bold, italic, fast blink, and concealed are not supported on all terminals. To set the video attributes half-intensity and underline, specify the following:

```
@(-35,10)
```

In this example, 10 is an additive key composed of 2 (half-intensity) plus 8 (underline).

Set Color Pair @(IT$SCOLPR)

Sets the background and foreground colors to a combination that you have previously defined in your system *terminfo* file.

Set Foreground Color @(IT$FCOLOR)

Sets the color that is used to display characters on the screen. @(IT$FCOLOR, *arg*) always takes an argument that specifies the foreground color to be chosen, as follows:

Value	Name	Description
0	IT$63	Black
1	IT$RED	Red
2	IT$GREEN	Green
3	IT$YELLOW	Yellow
4	IT$BLUE	Blue
5	IT$MAGENTA	Magenta
6	IT$CYAN	Cyan
7	IT$WHITE	White
8	IT$DARK.RED	Dark red
9	IT$CERISE	Cerise
10	IT$ORANGE	Orange
11	IT$PINK	Pink
12	IT$DARK.GREEN	Dark green
13	IT$SEA.GREEN	Sea green
14	IT$LIME.GREEN	Lime green
15	IT$PALE.GREEN	Pale green
16	IT$BROWN	Brown
17	IT$CREAM	Cream
18	IT$DARK.BLUE	Dark blue
19	IT$SLATE.BLUE	Slate blue
20	IT$VIOLET	Violet
21	IT$PALE.BLUE	Pale blue
22	IT$PURPLE	Purple
23	IT$PLUM	Plum

Value	Name	Description (Continued)
24	IT$DARK.CYAN	Dark cyan
25	IT$SKY.BLUE	Sky blue
26	IT$GREY	Grey

Note that the color attributes are not additive. Only one foreground color at a time can be displayed. If a terminal does not support a particular color, a request for that color should return an empty string.

Set Background Color @(IT$BCOLOR)

Sets the background color that is used to display characters on the screen. The @(IT$BCOLOR, *arg*) function always has an argument that specifies the background color to be chosen. (See "Set Foreground Color @(IT$FCOLOR)" for a list of available colors.)

Start Line Graphics @(IT$SLINEGRFX)

Switches on the line graphics mode for drawing boxes or lines on the screen.

End Line Graphics @(IT$ELINEGRFX)

Switches off the line graphics mode.

Line Graphics Character @(IT$LINEGRFXCH)

Specifies the line graphics character required. The argument can be one of the following tokens:

Arguments for Line Graphics Characters

Token	Value	Description
IT$GRFX.CROSS	0	Cross piece
IT$GRFX.H.LINE	1	Horizontal line
IT$GRFX.V.LINE	2	Vertical line
IT$GRFX.TL.CORNER	3	Top-left corner
IT$GRFX.TR.CORNER	4	Top-right corner
IT$GRFX.BL.CORNER	5	Bottom-left corner

Arguments for Line Graphics Characters (Continued)

Token	Value	Description
IT$GRFX.BR.CORNER	6	Bottom-right corner
IT$GRFX.TOP.TEE	7	Top-edge tee piece
IT$GRFX.LEFT.TEE	8	Left-edge tee piece
IT$GRFX.RIGHT.TEE	9	Right-edge tee piece
IT$GRFX.BOTTOM.TEE	10	Bottom-edge tee piece

Disable Manual Input @(IT$DMI)

Locks the terminal's keyboard.

Enable Manual Input @(IT$EMI)

Unlocks the terminal's keyboard.

Blank Screen @(IT$BSCN)

Blanks the terminal's display. Subsequent output to the screen is not visible until the unblank screen function, @(IT$UBS), is used.

Unblank Screen @(IT$UBS)

Restores the terminal's display after it was blanked. The previous contents of the screen, and any subsequent updates, become visible.

Scroll Up @(IT$SU)

Moves the entire contents of the display up one line. For *m* greater than 0, the function @(IT$SU, *m*) moves the display up *m* lines or until the bottom of the display is reached, whichever occurs first. For each line that is scrolled, the first line is removed from sight and another line is moved into the last line. This function works only if the terminal is capable of addressing character positions that do not all fit on the screen, such that some lines are not displayed. This normally requires the terminal to be set to vertical two-page mode in the initialization string. The effect of attempting to scroll the terminal too far is undefined.

Scroll Down @(IT$SD)

Moves the entire contents of the display down one line. For *m* greater than 0, the function @(IT$SD, *m*) moves the display down *m* lines or until the top of the display is reached, whichever occurs first. For each line that is scrolled, the last line is removed from sight and another line is moved into the top line. This function works only if the terminal is capable of addressing character positions that do not all fit on the screen, such that some lines are not displayed. This normally requires the terminal to be set to vertical two-page mode in the initialization string. The effect of attempting to scroll the terminal too far is undefined.

Scroll Right @(IT$SR)

Moves the entire contents of the display one column to the right. For *m* greater than 0, the function @(IT$SR, *m*) moves the display *m* columns to the right or until the left edge of the display is reached, whichever occurs first. For each column scrolled, the rightmost column is removed from sight and another leftmost column appears. This function works only if the terminal is capable of addressing character positions that do not fit on the screen, such that some columns are not displayed. This normally requires the terminal to be set to horizontal two-page mode in the initialization string. The effect of attempting to scroll the terminal too far is undefined.

Scroll Left @(IT$SL)

Moves the entire contents of the display one column to the left. For *m* greater than 0, the function @(IT$SL, *m*) moves the display *m* columns to the left or until the right edge of the display is reached, whichever happens first. For each column scrolled, the leftmost column is removed from sight and another rightmost column appears. This function works only if the terminal is capable of addressing character positions that do not fit on the screen, such that some columns are not displayed. This normally requires the terminal to be set to horizontal two-page mode in the initialization string. The effect of attempting to scroll the terminal too far is undefined.

Set Line Truncate @(IT$SLT)

Makes the cursor stay in the last position on the line when characters are printed past the last position.

Reset Line Truncate @(IT$RLT)

Makes the cursor move to the first position on the next line down when characters are printed past the last position.

Set Numeric Keypad @(IT$SNK)

Sets keys on the numeric keypad to the labelled functions instead of numbers.

Reset Numeric Keypad @(IT$RNK)

Resets keys on the numeric keypad to numbers.

Start Bold @(IT$SBOLD)

Starts bold mode; subsequently, any characters entered are shown more brightly on the screen.

End Bold @(IT$EBOLD)

Ends bold mode; characters revert to normal screen brightness.

Start Secure Mode @(IT$SSECUR)

Characters entered in this setting are not shown on the screen. This function can be used when entering passwords, for example.

End Secure Mode @(IT$ESECURE)

Switches off secure mode; characters appear on screen.

Start Screen Protect Mode @(IT$SSCRPROT)

Switches on start protect mode. Characters entered in this mode are not removed when the screen is cleared.

End Screen Protect Mode @(IT$ESCRPROT)

Switches off screen protect mode.

System Line Display @(IT$SLD)

Redisplays the user-defined characters that were sent by the system line set function, @(IT$SLS). The system line is defined as an extra line on the terminal display but is addressable by the normal cursor positioning sequence. On most terminals

the system line normally contains a terminal status description. The number of usable lines on the screen does not change.

System Line Reset @(IT$SLR)

Removes from the display the characters that were set by the @(IT$SLS) function and replaces them with the default system status line. The number of usable lines on the screen does not change.

System Line Set @(IT$SLS)

Displays the user-defined status line, and positions the cursor at the first column of the status line. Subsequent printing characters sent to the terminal are displayed on the status line. Issuing a system line reset function, @(IT$SLR), terminates printing on the status line, and leaves the cursor position undefined. The characters printed between the issuing of @(IT$SLS) and @(IT$SLR) can be recalled subsequently and displayed on the line by issuing an @(IT$SLD) function.

Cursor Horizontal Absolute @(IT$CHA)

Positions the cursor at column *m* of the current line. If *m* is omitted, the default is 0. The @(IT$CHA, *m*) function must have the same effect as @(*m*).

Erase Character @(IT$ECH)

Erases the character under the cursor and replaces it with one or more spaces, determined by the argument *m*. If you do not specify *m*, or you specify a value for *m* that is less than 2, only the character under the cursor is replaced. If you specify an argument whose value is greater than 1, the function replaces the character under the cursor, and *m* –1 characters to the right of the cursor, with spaces. The cursor position is unchanged.

IT$NPC, IT$DISPLAY, and IT$MINIBUF

Reserved for EDFS attributes.

Lock Line @(IT$LOKL)

Locks line *n* of the screen display (top line is 0). The line cannot be modified, moved, or deleted from the screen until it is unlocked.

Unlock Line @(IT$UNLL)

Unlocks line *n* of the screen display allowing it to be modified, moved, or deleted.

Display Marks @(IT$MARKSUBS)

Returns the characters used to display UniVerse delimiters on screen. From left to right, the delimiters are: item, field, value, subvalue, and text.

Allocated for VMARK@(–80) to @(–100)

These functions are reserved for VMARK.

Allocated for General Use @(–101) to @(–128)

These functions are available for any additional terminal definition strings that you require.

Video Attributes: Points to Note

Terminals whose video attributes are described as embedded or on-screen use a character position on the terminal screen whenever a start or stop video attribute is received. Programs driving such terminals must not change an attribute in the middle of a contiguous piece of text. You must leave at least one blank character position at the point where the attribute changes. The field in the terminal definition record called *xmc* is used to specify the number of character positions required for video attributes. A program can examine this field, and take appropriate action. To do this, the program must execute GET.TERM.TYPE and examine the @SYSTEM-.RETURN.CODE variable, or use the definition VIDEO.SPACES from the TERM INFO.H file.

Many terminals do not clear video attributes automatically when the data on a line is cleared or deleted. The recommended programming practice is to reposition to the point at which a start attribute was emitted, and overwrite it with an end attribute, before clearing the line.

On some terminals you can set up the Clear to End of Line sequence to clear both data and video attributes. This is done by combining the strings for erase data from active position to end of line, selecting Graphic Rendition normal, and changing all video attributes from active position to end of line. Sending the result of the

@(IT$CLEOL) function causes both the visible data on the line to be cleared, and all video attributes to be set to normal, after the cursor position.

Note: Where possible, you should try to ensure that any sequences that clear data also clear video attributes. Note that this may not be the case for all terminal types.

An exception is @(IT$CS) clear screen. The sequence associated with this function should always clear not only all data on the screen but also reset any video attributes to normal.

Examples

The following example displays “Demonstration” at column 5, line 20:

```
PRINT @(5,20):"Demonstration"
```

In the following example, the PRINT statement positions the cursor to home, at the top-left corner of the screen, and clears the screen:

```
PRINT @(IT$CS):
```

The $INCLUDE statement is used to include the ATFUNCTIONS insert file of equate names. Assignment statements are used to assign the evaluated @ functions to variables. The variables are used in PRINT statements to produce code that clears the screen and returns the cursor to its original position; positions the cursor at column 5, line 20; turns on the reverse video mode; prints the string; and turns off the reverse video mode.

```
$INCLUDE UNIVERSE.INCLUDE ATFUNCTIONS.H
CLS = @(IT$CS)
REVERSE.ON = @(IT$SREV)
REVERSE.OFF = @(IT$EREV)
.
.
.
PRINT CLS: @(5,20):
PRINT REVERSE.ON:"THIS IS REVERSE VIDEO":REVERSE.OFF
```

The next example displays any following text in yellow letters:

```
PRINT @(IT$FCOLOR, IT$YELLOW)
```

The next example displays any following text on a cyan background:

```
PRINT @(IT$BCOLOR, IT$CYAN)
```

The next example gives a yellow foreground, not a green foreground, because color changes are not additive:

```
PRINT @(IT$FCOLOR, IT$BLUE):@(IT$FCOLOR, IT$YELLOW)
```

If you have a terminal that supports colored letters on a colored background, the next example displays the text “Hello” in yellow on a cyan background. All subsequent output is in yellow on cyan until another color @ function is used. If your color terminal cannot display colored foreground on colored background, only the last color command is used, so that this example displays the text “Hello” in yellow on a black background.

```
PRINT @(IT$BCOLOR,IT$CYAN):@(IT$FCOLOR,IT$YELLOW):"Hello"
```

If your color terminal cannot display colored foreground on colored background, the previous example displays the text “Hello” in black on a cyan background.

The next example gives the same result as the previous example for a terminal that supports colored letters on a colored background. Strings containing the @ functions can be interpreted as a sequence of instructions, which can be stored for subsequent frequent reexecution.

```
PRINT @(IT$FCOLOR,IT$YELLOW):@(IT$BCOLOR,IT$CYAN):"Hello"
```

In the last example, the screen is cleared, the cursor is positioned to the tenth column in the tenth line, and the text “Hello” is displayed in foreground color cyan. The foreground color is then changed to white for subsequent output. This sequence of display instructions can be executed again, whenever it is required, by a further PRINT SCREEN statement.

```
SCREEN = @(IT$CS):@(10,10):@(IT$FCOLOR,IT$CYAN):"Hello"
SCREEN = SCREEN:@(IT$FCOLOR,IT$WHITE)
PRINT SCREEN
```

Syntax

expression **[** [*start*,] *length* **]**

expression **[** *delimiter*, *occurrence*, *fields* **]**

Description

Use the **[ ]** operator (square brackets) to extract a substring from a character string. The larger bold brackets are part of the syntax and must be typed.

expression evaluates to any character string.

start is an expression that evaluates to the starting character position of the substring. If *start* is 0 or a negative number, the starting position is assumed to be 1. If you omit *start*, the starting position is calculated according to the following formula:

string.length – *substring.length* + 1

This lets you specify a substring consisting of the last *n* characters of a string without having to calculate the string length.

If *start* exceeds the number of characters in *expression*, an empty string results. An empty string also results if *length* is 0 or a negative number. If the sum of *start* and *length* exceeds the number of characters in the string, the substring ends with the last character of the string.

length is an expression that evaluates to the length of the substring.

Use the second syntax to return a substring located between the specified number of occurrences of the specified delimiter. This syntax performs the same function as the FIELD function.

delimiter can be any string, including field mark, value mark, and subvalue mark characters. It delimits the start and end of the substring (all that appears within the two delimiters). If *delimiter* consists of more than one character, only the first character is used.

occurrence specifies which occurrence of the delimiter is to be used as a terminator. If *occurrence* is less than 1, 1 is assumed.

fields specifies the number of successive fields after the delimiter specified by *occurrence* that are to be returned with the substring. If the value of *fields* is less than 1, 1 is assumed. The delimiter is part of the returned value in the successive fields.

If the delimiter or the occurrence specified does not exist within the string, an empty string is returned. If *occurrence* specifies 1 and no delimiter is found, the entire string is returned.

If *expression* is the null value, any substring extracted from it will also be the null value.

Examples

In the following example (using the second syntax) the fourth # is the terminator of the substring to be extracted, and one field is extracted:

```
A="###DHHH#KK"
PRINT A["#",4,1]
```

This is the result:

```
DHHH
```

The following syntaxes specify substrings that start at character position 1:

expression **[** 0, *length* **]**

expression **[** –1, *length* **]**

The following example specifies a substring of the last five characters:

```
"1234567890" [5]
```

This is the result:

```
67890
```

All substring syntaxes can be used in conjunction with the assignment operator (=). The new value assigned to the variable replaces the substring specified by the **[ ]** operator. For example:

```
A='12345'
A[3]=1212
PRINT "A=",A
```

returns the following:

```
A= 121212
```

A[3] replaces the last three characters of A (345) with the newly assigned value for that substring (1212).

The FIELDSTORE function provides the same functionality as assigning the three-argument syntax of the **[ ]** operator.

Syntax

ABORT [*expression* ...]

ABORTE [*expression* ...]

ABORTM [*expression* ...]

Description

Use the ABORT statement to terminate execution of a BASIC program and return to the UniVerse prompt. ABORT differs from STOP in that a STOP statement returns to the calling environment (for example, a menu, a paragraph, another BASIC program following an EXECUTE statement, and so on), whereas ABORT terminates all calling environments as well as the BASIC program. You can use it as part of an IF...THEN statement to terminate processing if certain conditions exist.

If *expression* is used, it is printed when the program terminates. If *expression* evaluates to the null value, nothing is printed.

The ABORTE statement is the same as the ABORT statement except that it behaves as if $OPTIONS STOP.MSG were in force. This causes ABORT to use the ERRMSG file to produce error messages instead of using the specified text. If *expression* in the ABORTE statement evaluates to the null value, the default error message is printed:

```
Message ID is NULL:  undefined error
```

For information about the ERRMSG file, see the ERRMSG statement.

The ABORTM statement is the same as the ABORT statement except that it behaves as if $OPTIONS –STOP.MSG were in force. This causes ABORT to use the specified text instead of text from the ERRMSG file.

Example

```
PRINT "DO YOU WANT TO CONTINUE?":
INPUT A
IF A="NO" THEN ABORT
```

This is the program output:

```
DO YOU WANT TO CONTINUE?NO
Program "TEST": Line 3, Abort.
```

ABS function

Syntax

ABS (*expression*)

Description

Use the ABS function to return the absolute value of any numeric expression. The absolute value of an expression is its unsigned magnitude. If *expression* is negative, the value returned is:

–*expression*

For example, the absolute value of –6 is 6.

If *expression* is positive, the value of *expression* is returned. If *expression* evaluates to the null value, null is returned.

Example

```
Y = 100
X = ABS(43-Y)
PRINT X
```

This is the program output:

```
57
```

Syntax

ABSS (*dynamic.array*)

Description

Use the ABSS function to return the absolute values of all the elements in a dynamic array. If an element in *dynamic.array* is the null value, null is returned for that element.

Example

```
Y = REUSE(300)
Z = 500:@VM:400:@VM:300:@SM:200:@SM:100
A = SUBS(Z,Y)
PRINT A
PRINT ABSS(A)
```

This is the program output:

```
200v100v0s-100s-200
200v100v0s100s200
```

ACOS function

Syntax

ACOS (*expression*)

Description

Use the ACOS function to return the trigonometric arc-cosine of *expression*. *expression* must be a numeric value. The result is expressed in degrees. If *expression* evaluates to the null value, null is returned. The ACOS function is the inverse of the COS function.

Example

```
PRECISION 5
PRINT "ACOS(0.707106781) = ":ACOS(0.707106781):" degrees"
```

This is the program output:

```
ACOS(0.707106781) = 45 degrees
```

Syntax

ADDS (*array1*, *array2*)

CALL –ADDS (*return.array*, *array1*, *array2*)

CALL !ADDS (*return.array*, *array1*, *array2*)

Description

Use the ADDS function to create a dynamic array of the element-by-element addition of two dynamic arrays.

Each element of *array1* is added to the corresponding element of *array2*. The result is returned in the corresponding element of a new dynamic array. If an element of one array has no corresponding element in the other array, the existing element is returned. If an element of one array is the null value, null is returned for the sum of the corresponding elements.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A = 2:@VM:4:@VM:6:@SM:10
B = 1:@VM:2:@VM:3:@VM:4
PRINT ADDS(A,B)
```

This is the program output:

```
3v6v9s10v4
```

ALPHA function

Syntax

ALPHA (*expression*)

Description

Use the ALPHA function to determine whether *expression* is an alphabetic or nonalphabetic string. If *expression* contains the characters a through z or A through Z, it evaluates to true and a value of 1 is returned. If *expression* contains any other character or an empty string, it evaluates to false and a value of 0 is returned. If *expression* evaluates to the null value, null is returned.

Example

```
PRINT "ALPHA('ABCDEFG') = ":ALPHA('ABCDEFG')
PRINT "ALPHA('abcdefg') = ":ALPHA('abcdefg')
PRINT "ALPHA('ABCDEFG.') = ":ALPHA('ABCDEFG.')
PRINT "ALPHA('SEE DICK') = ":ALPHA('SEE DICK')
PRINT "ALPHA('4 SCORE') = ":ALPHA('4 SCORE')
PRINT "ALPHA('') = ":ALPHA('')
```

This is the program output:

```
ALPHA('ABCDEFG') = 1
ALPHA('abcdefg') = 1
ALPHA('ABCDEFG.') = 0
ALPHA('SEE DICK') = 0
ALPHA('4 SCORE') = 0
ALPHA('') = 0
```

Syntax

ANDS (*array1*, *array2*)

CALL –ANDS (*return.array*, *array1*, *array2*)

CALL !ANDS (*return.array*, *array1*, *array2*)

Description

Use the ANDS function to create a dynamic array of the logical AND of corresponding elements of two dynamic arrays.

Each element of the new dynamic array is the logical AND of the corresponding elements of *array1* and *array2*. If an element of one dynamic array has no corresponding element in the other dynamic array, a false (0) is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

If both corresponding elements of *array1* and *array2* are the null value, null is returned for those elements. If one element is the null value and the other is 0 or an empty string, a false is returned for those elements.

Example

```
A = 1:@SM:4:@VM:4:@SM:1
B = 1:@SM:1-1:@VM:2
PRINT ANDS(A,B)
```

This is the program output:

```
1s0v1s0
```

ASCII function

Syntax

ASCII (*expression*)

Description

Use the ASCII function to convert each character of *expression* from its EBCDIC representation value to its ASCII representation value. If *expression* evaluates to the null value, null is returned.

The ASCII function and the EBCDIC function perform complementary operations.

Example

```
X = EBCDIC('ABC 123')
Y = ASCII(X)
PRINT "EBCDIC", "ASCII", " Y "
PRINT "------", "-----", "---"
FOR I = 1 TO LEN (X)
PRINT SEQ(X[I,1]) , SEQ(Y[I,1]),Y[I,1]
NEXT I
```

This is the program output:

```
EBCDIC   ASCII      Y
------   -----     ---
193      65        A
194      66        B
195      67        C
64       32
241      49        1
242      50        2
243      51        3
```

Syntax

ASIN (*expression*)

Description

Use the ASIN function to return the trigonometric arc-sine of *expression*. The *expression* must be a numeric value. The result is expressed in degrees. If *expression* evaluates to the null value, null is returned. The ASIN function is the inverse of the SIN function.

Example

```
PRECISION 5
PRINT "ASIN(0.707106781) = ":ASIN(0.707106781):" degrees"
```

This is the program output:

```
ASIN(0.707106781) = 45 degrees
```

ASSIGNED function

Syntax

ASSIGNED (*variable*)

Description

Use the ASSIGNED function to determine if *variable* is assigned a value. ASSIGNED returns 1 (true) if *variable* is assigned a value, including common variables and the null value. It returns 0 (false) if *variable* is not assigned a value.

PICK Flavor

When you run UniVerse in a PICK flavor account, all common variables are initially unassigned. ASSIGNED returns 0 (false) for common variables until the program explicitly assigns them a value.

Example

```
A = "15 STATE STREET"
C = 23
X = ASSIGNED(A)
Y = ASSIGNED(B)
Z = ASSIGNED(C)
PRINT X,Y,Z
```

This is the program output:

```
1  0  1
```

Syntax

variable = *expression*

variable += *expression*

variable –= *expression*

variable := *expression*

Description

Use assignment statements to assign a value to a variable. The variable can be currently unassigned (that is, one that has not been assigned a value by an assignment statement, a READ statement, or any other statement that assigns values to variables) or have an old value that is to be replaced. The assigned value can be a constant or an expression. It can be any data type (that is, numeric, character string, or the null value).

Use the operators += , –= , and := to alter the value of a variable. The += operator adds the value of *expression* to *variable*. The –= operator subtracts the value of *expression* from *variable*. The := operator concatenates the value of *expression* to the end of *variable*.

Use the system variable @NULL to assign the null value to a variable:

variable = @NULL

Use the system variable @NULL.STR to assign a character string containing only the null value (more accurately, the character used to represent the null value) to a variable:

variable = @NULL.STR

Example

```
EMPL=86
A="22 STAGECOACH LANE"
X='$4,325'
B=999
PRINT "A= ":A,"B= ":B,"EMPL= ":EMPL
B+=1
PRINT "X= ":X,"B= ":B
```

This is the program output:

```
A= 22 STAGECOACH LANE   B= 999 EMPL= 86
X= $4,325   B= 1000
```

Syntax

ATAN (*expression*)

Description

Use the ATAN function to return the trigonometric arc-tangent of *expression*. *expression* must be a numeric value. The result is expressed in degrees. If *expression* evaluates to the null value, null is returned. The ATAN function is the inverse of the TAN function.

Examples

The following example prints the numeric value 135 and the angle, in degrees, that has an arc-tangent of 135:

```
PRINT 135, ATAN(135)
```

The next example finds what angle has an arc-tangent of 1:

```
X = ATAN(1)
PRINT 1, X
```

This is the program output:

```
135       89.5756
1         45
```

AUTHORIZATION statement

Syntax

AUTHORIZATION "*username*"

Description

Use the AUTHORIZATION statement to specify or change the effective run-time user of a program. After an AUTHORIZATION statement is executed, any SQL security checking acts as if *username* is running the program.

username is a valid login name on the machine where the program is run. *username* must be a constant. *username* is compiled as a character string whose user identification (UID) number is looked up in the */etc/passwd* file at run time.

If your program accesses remote files across UV/Net, *username* must also be a valid login name on the remote machine.

An AUTHORIZATION statement changes only the user name that is used for SQL security checking while the program is running. It does not change the actual user name, nor does it change the user's effective UID at the operating system level. If a program does not include an AUTHORIZATION statement, it runs with the user name of the user who invokes it.

You can change the effective user of a program as many times as you like. The *username* specified by the most recently executed AUTHORIZATION statement remains in effect for subsequent EXECUTE and PERFORM statements as well as for subroutines.

When a file is opened, the effective user's permissions are stored in the file variable. These permissions apply whenever the file variable is referenced, even if a subsequent AUTHORIZATION statement changes the effective user name.

The effective user name is stored in the system variable @AUTHORIZATION.

A program using the AUTHORIZATION statement must be compiled on the machine where the program is to run. To compile the AUTHORIZATION statement, SQL DBA privilege is required. If the user compiling the program does not have DBA privilege, the program will not be compiled. You cannot run the program on a machine different from the one where it was compiled. If you try, the program terminates with a fatal error message.

Example

```
AUTHORIZATION "susan"
OPEN "","SUES.FILE" TO FILE.S ELSE PRINT "CAN'T OPEN SUES.FILE"
AUTHORIZATION "bill"
OPEN "","BILLS.FILE" TO FILE.B ELSE PRINT "CAN'T OPEN
BILLS.FILE"
FOR ID = 5000 TO 6000
   READ SUE.ID FROM FILE.S, ID THEN PRINT ID ELSE NULL
   READ BILL.ID FROM FILE.B, ID THEN PRINT ID ELSE NULL
NEXT ID
```

BEGIN CASE statement

Use the BEGIN CASE statement to begin a set of CASE statements. For details, see the CASE statement.

Syntax

BEGIN TRANSACTION [ISOLATION LEVEL *level*]
[*statements*]

Description

Use the BEGIN TRANSACTION statement to indicate the beginning of a transaction.

The ISOLATION LEVEL clause sets the transaction isolation level for the duration of that transaction. The isolation level reverts to the original value at the end of the transaction.

level can be one of the following keywords or an integer from 0 through 4:

Effects of ISOLATION LEVELs

Integer	Keyword	Effect on This Transaction
0	NO.ISOLATION	Prevents lost updates.[1]
1	READ.UNCOMMITTED	Prevents lost updates.
2	READ.COMMITTED	Prevents lost updates and dirty reads.
3	REPEATABLE.READ	Prevents lost updates, dirty reads, and nonrepeatable reads.
4	SERIALIZABLE	Prevents lost updates, dirty reads, nonrepeatable reads, and phantom writes.

[1]List updates are prevented if the ISOMODE configurable parameter is set to 1 or 2.

Examples

The following examples both start a transaction at isolation level 3:

```
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE.READ
BEGIN TRANSACTION ISOLATION LEVEL 3
```

BITAND function

Syntax

BITAND (*expression1*, *expression2*)

Description

Use the BITAND function to perform the bitwise AND comparison of two integers specified by numeric expressions. The bitwise AND operation compares two integers bit by bit. It returns a bit of 1 if both bits are 1; otherwise it returns a bit of 0.

If either *expression1* or *expression2* evaluates to the null value, null is returned.

Noninteger values are truncated before the operation is performed.

The BITAND operation is performed on a 32-bit twos-complement word.

Note: Differences in hardware architecture can make the use of the high-order bit nonportable.

Example

```
PRINT BITAND(6,12)
* The binary value of 6  =  0110
* The binary value of 12 =  1100
```

This results in 0100, and the following output is displayed:

```
4
```

Syntax

BITNOT (*expression* [,*bit#*])

Description

Use the BITNOT function to return the bitwise negation of an integer specified by any numeric expression.

bit# is an expression that evaluates to the number of the bit to invert. If *bit#* is unspecified, BITNOT inverts each bit. It changes each bit of 1 to a bit of 0 and each bit of 0 to a bit of 1. This is equivalent to returning a value equal to the following:

(–*expression*)–1

If *expression* evaluates to the null value, null is returned. If *bit#* evaluates to the null value, the BITNOT function fails, the program terminates, and a run-time error message is produced.

Noninteger values are truncated before the operation is performed.

The BITNOT operation is performed on a 32-bit twos-complement word.

Note: Differences in hardware architecture can make the use of the high-order bit nonportable.

Example

```
PRINT BITNOT(6),BITNOT(15,0),BITNOT(15,1),BITNOT(15,2)
```

This is the program output:

```
–7 14 13 11
```

BITOR function

Syntax

BITOR (*expression1*, *expression2*)

Description

Use the BITOR function to perform the bitwise OR comparison of two integers specified by numeric expressions. The bitwise OR operation compares two integers bit by bit. It returns the bit 1 if the bit in either or both numbers is 1; otherwise it returns the bit 0.

If either *expression1* or *expression2* evaluates to the null value, null is returned.

Noninteger values are truncated before the operation is performed.

The BITOR operation is performed on a 32-bit twos-complement word.

Note: Differences in hardware architecture can make the use of the high-order bit nonportable.

Example

```
PRINT BITOR(6,12)
* Binary value of 6  = 0110
* Binary value of 12 = 1100
```

This results in 1110, and the following output is displayed:

```
14
```

Syntax

BITRESET (*expression*, *bit#*)

Description

Use the BITRESET function to reset to 0 the bit number of the integer specified by *expression*. Bits are counted from right to left. The number of the rightmost bit is 0. If the bit is 0, it is left unchanged.

If *expression* evaluates to the null value, null is returned. If *bit#* evaluates to the null value, the BITRESET function fails, the program terminates, and a run-time error message is produced.

Noninteger values are truncated before the operation is performed.

Example

```
PRINT BITRESET(29,0),BITRESET(29,3)
* The binary value of 29 = 11101
* The binary value of 28 = 11100
* The binary value of 21 = 10101

PRINT BITRESET(2,1),BITRESET(2,0)
* The binary value of 2 = 10
* The binary value of 0 =  0
```

This is the program output:

```
28  21
0   2
```

BITSET function

Syntax

BITSET (*expression*, *bit#*)

Description

Use the BITSET function to set to 1 the bit number of the integer specified by *expression*. The number of the rightmost bit is 0. If the bit is 1, it is left unchanged.

If *expression* evaluates to the null value, null is returned. If *bit#* evaluates to the null value, the BITSET function fails, the program terminates, and a run-time error message is produced.

Noninteger values are truncated before the operation is performed.

Example

```
PRINT BITSET(20,0),BITSET(20,3)
* The binary value of 20 = 10100
* The binary value of 21 = 10101
* The binary value of 28 = 11100

PRINT BITSET(2,0),BITSET(2,1)
* The binary value of 2 = 10
* The binary value of 3 = 11
```

This is the program output:

```
21  28
3   2
```

Syntax

BITTEST (*expression*, *bit#*)

Description

Use the BITTEST function to test the bit number of the integer specified by *expression*. The function returns 1 if the bit is set; it returns 0 if it is not. Bits are counted from right to left. The number of the rightmost bit is 0.

If *expression* evaluates to the null value, null is returned. If *bit#* evaluates to null, the BITTEST function fails, the program terminates, and a run-time error message is produced.

Noninteger values are truncated before the operation is performed.

Example

```
PRINT BITTEST(11,0),BITTEST(11,1),BITTEST(11,2),BITTEST(11,3)
* The binary value of 11 = 1011
```

This is the program output:

```
1  1  0  1
```

BITXOR function

Syntax

BITXOR (*expression1*, *expression2*)

Description

Use the BITXOR function to perform the bitwise XOR comparison of two integers specified by numeric expressions. The bitwise XOR operation compares two integers bit by bit. It returns a bit 1 if only one of the two bits is 1; otherwise it returns a bit 0.

If either *expression1* or *expression2* evaluates to the null value, null is returned.

Noninteger values are truncated before the operation is performed.

The BITXOR operation is performed on a 32-bit twos-complement word.

Note: Differences in hardware architecture can make the use of the high-order bit nonportable.

Example

```
PRINT BITXOR(6,12)
* Binary value of 6  = 0110
* Binary value of 12 = 1100
```

This results in 1010, and the following output is displayed:

```
10
```

Syntax

BREAK [KEY] ON

BREAK [KEY] OFF

BREAK [KEY] *expression*

Description

Use the BREAK statement to enable or disable the **Intr**, **Quit**, and **Susp** keys on the keyboard.

When the BREAK ON statement is in effect, pressing **Intr**, **Quit**, or **Susp** causes operations to pause.

When the BREAK OFF statement is in effect, pressing **Intr**, **Quit**, or **Susp** has no effect. This prevents a break in execution of programs that you do not want interrupted.

When *expression* is used with the BREAK statement, the value of *expression* determines the status of the **Intr**, **Quit**, and **Susp** keys. If *expression* evaluates to false (0, an empty string, or the null value), the **Intr**, **Quit**, and **Susp** keys are disabled. If *expression* evaluates to true (not 0, an empty string, or the null value), the **Intr**, **Quit**, and **Susp** keys are enabled.

A counter is maintained for the BREAK statement. It counts the number of executed BREAK ON and BREAK OFF commands. When program control branches to a subroutine, the value of the counter is maintained; it is not set back to 0. For each BREAK ON statement executed, the counter decrements by 1; for each BREAK OFF statement executed, the counter increments by 1. The counter cannot go below 0. The **Intr**, **Quit**, and **Susp** keys are enabled only when the value of the counter is 0. The following example illustrates the point:

Counters and the BREAK Statement

Statement from	Command	Counter	Key Status
-	-	0	ON
Main program	BREAK OFF	+1	OFF
Subroutine	BREAK OFF	+2	OFF
Subroutine	BREAK ON	+1	OFF
Main program	BREAK ON	0	ON

Examples

The following example increases the counter by 1:

```
BREAK KEY OFF
```

The following example decreases the counter by 1:

```
BREAK KEY ON
```

The following example disables the **Intr**, **Quit**, and **Susp** keys if QTY is false, 0, an empty string, or the null value; it enables them if QTY is true, not 0, not an empty string, or not the null value:

```
BREAK QTY ;*
```

Syntax

BSCAN *ID.variable*[,*rec.variable*] [FROM [[*file.variable*,] *record*]
[USING *indexname*] [RESET] [BY *seq*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the BSCAN statement to scan the leaf nodes of a B-tree file (type 25) or of a secondary index. The record ID returned by the current scan operation is assigned to *ID.variable*. If you specify *rec.variable*, the contents of the record whose ID is *ID.variable* is assigned to it.

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement). If the file is neither accessible nor open, the program terminates and a run-time error message is produced.

record is an expression that evaluates to a record ID of a record in the B-tree file. If the USING clause is used, *record* is a value in the specified index. *record* specifies the relative starting position of the scan.

record need not exactly match an existing record ID or value. If it does not, the scan finds the next or previous record ID or value, depending on whether the scan is in ascending or descending order. For example, depending on how precisely you want to specify the starting point at or near the record ID or value SMITH, *record* can evaluate to SMITH, SMIT, SMI, SM, or S.

If you do not specify *record*, the scan starts at the leftmost slot of the leftmost leaf, or the rightmost slot of the rightmost leaf, depending on the value of the *seq* expression. The scan then moves in the direction specified in the BY clause.

indexname is an expression that evaluates to the name of a secondary index associated with the file.

RESET resets the internal B-tree scan pointer. If the scanning order is ascending, the pointer is set to the leftmost slot of the leftmost leaf; if the order is descending, the pointer is set to the rightmost slot of the rightmost leaf. If you do not specify *seq*, the scan is done in ascending order. If you specify *record* in the FROM clause, RESET is ignored.

seq is an expression that evaluates to A or D; it specifies the direction of the scan. "A", the default, specifies ascending order. "D" specifies descending order.

If the BSCAN statement finds a valid record ID, or a record ID and its associated data, the THEN statements are executed; the ELSE statements are ignored. If the scan does not find a valid record ID, or if some other error occurs, any THEN statements are ignored, and the ELSE statements are executed.

Any file updates executed in a transaction (that is, between a BEGIN TRANSACTION statement and a COMMIT statement) are not accessible to the BSCAN statement until after the COMMIT statement has been executed.

The STATUS function returns the following values after the BSCAN statement is executed:

0 The scan proceeded beyond the leftmost or rightmost leaf node. *ID.variable* and *rec.variable* are set to empty strings.

1 The scan returned an existing record ID, or a record ID that exactly matches the record ID specified by *record*.

2 The scan returned a record ID that does not exactly match *record*. *ID.variable* is either the next or the previous record ID in the B-tree, depending on the direction of the scan.

3 The file is not a B-tree (type 25) file, or, if the USING clause is used, the file has no active secondary indices.

4 *indexname* does not exist.

5 *seq* does not evaluate to A or D.

6 The index specified by *indexname* needs to be built.

10 An internal error was detected.

Example

The following example shows how you might indicate that the ELSE statements were executed because the contents of the leaf nodes were exhausted:

```
BSCAN ID,REC FROM FILE,MATCH USING "PRODUCT" BY "A" THEN
   PRINT ID,REC
END ELSE
   ERR = STATUS()
   BEGIN CASE
      CASE ERR = 0
         PRINT "Exhausted leaf node contents."
      CASE ERR = 3
         PRINT "No active indices, or file is not type 25."
```

```
        CASE ERR = 4
           PRINT "Index name does not exist."
        CASE ERR = 5
           PRINT "Invalid BY clause value."
        CASE ERR = 6
           PRINT "Index must be built."
        CASE ERR = 10
           PRINT "Internal error detected."
     END CASE
     GOTO EXIT.PROGRAM:
  END
```

CALL statement

Syntax

CALL *subroutine* [([MAT] *argument* [, [MAT] *argument* ...])]

variable = '*subroutine*'
CALL @*variable* [([MAT] *argument* [, [MAT] *argument* ...])]

Description

Use the CALL statement to transfer program control from the calling program to an external subroutine that has been compiled and cataloged.

Locally cataloged subroutines can be called directly. Specify *subroutine* using the exact name under which it was cataloged. For more details, see the CATALOG command in the *UniVerse BASIC Programmer's Guide*.

External subroutines can be called directly or indirectly. To call a subroutine indirectly, the name under which the subroutine is cataloged must be assigned to a variable or to an element of an array. This variable name or array element specifier, prefixed with an at sign (@), is used as the operand of the CALL statement.

The first time a CALL is executed, the system searches for the subroutine in a cataloged library and changes a variable that contains the subroutine name to contain its location information instead. This procedure eliminates the need to search the catalog again if the same subroutine is called later in the program. For indirect calls, the variable specified in the CALL as the @variable is used; for direct calls, an internal variable is used. With the indirect method, it is best to assign the subroutine name to the variable only once in the program, not every time the indirect CALL statement is used.

arguments are variables, arrays, array variables, expressions, or constants that represent actual values. You can pass one or more arguments from the calling program to a subroutine. The number of arguments passed in a CALL statement must equal the number of arguments specified in the SUBROUTINE statement that identifies the subroutine. If multiple arguments are passed, they must be separated by commas. If an argument requires more than one physical line, use a comma at the end of the line to indicate that the list continues.

If *argument* is an array, it must be preceded by the MAT keyword, and the array should be named and dimensioned in both the calling program and the subroutine before using this statement. If the array is not dimensioned in the subroutine, it must be declared using the MAT keyword in the SUBROUTINE statement. Other arguments can be passed at the same time regardless of the size of the array.

The actual values of *arguments* are not passed to the subroutine. Instead, a pointer to the location of each argument is passed. Passing a pointer instead of the values is more efficient when many values need to be passed to the subroutine. This method of passing arguments is called passing *by reference*; passing actual values is called passing *by value*.

All scalar and matrix variables are passed to subroutines by reference. If you want to pass variables by value, enclose them in parentheses. When data is passed by value, the contents of the variable in the main program do not change as a result of manipulations to the data in the subroutine. When data is passed by reference, the memory location of the variable is changed by manipulations in both the main program and the subroutines. Constants are passed to subroutines by value.

When an array is passed to an external subroutine as an argument in a CALL statement, any dimensions assigned to the array in the subroutine are ignored. The dimensions of the original array as it exists in the calling program are maintained. Therefore, it is a common and acceptable practice to dimension the array in the subroutine with subscripts or indices of one. For example, you could dimension the arrays in the subroutine as follows:

```
DIM A (1), B (1, 1), C (1, 1)
```

When the corresponding array arguments are passed from the calling program to the subroutine at run time, arrays A, B, and C inherit the dimensions of the arrays in the calling program. The indices in the DIM statement are ignored.

A better way to declare array arguments in a subroutine is to use the MAT keyword of the SUBROUTINE statement in the first line of the subroutine. The following example tells the subroutine to expect the three arrays A, B, and C:

```
SUBROUTINE X(MAT A, MAT B, MAT C)
```

When a RETURN statement is encountered in the subroutine, or when execution of the subroutine ends without encountering a RETURN statement, control returns to the statement following the CALL statement in the calling program. For more details, see the RETURN statement.

Examples

The following example calls the local subroutine SUB. It has no arguments.

```
CALL SUB
```

The following example calls the local subroutine QTY.ROUTINE with three arguments:

```
CALL QTY.ROUTINE(X,Y,Z)
```

The following example calls the subroutine cataloged as *PROGRAM.1 with six arguments. The argument list can be expressed on more than one line.

```
AAA="*PROGRAM.1"
CALL @AAA(QTY,SLS,ORDER,ANS,FILE.O,SEQ)
```

The following example calls the subroutine *MA with three arguments. Its index and three arguments are passed.

```
STATE.TAX(1,2)='*MA'
CALL @STATE.TAX(1,2)(EMP.NO,GROSS,NET)
```

The following example calls the subroutine cataloged as *SUB and two matrices are passed to two subroutine matrices. A third, scalar, argument is also passed.

```
GET.VALUE="*SUB"
DIM QTY(10)
DIM PRICE(10)
CALL @GET.VALUE( MAT QTY,MAT PRICE,COST )
```

The following example shows the SUBROUTINE statement in the subroutine SUB that is called by the preceding example. Note that the arrays Q and P need not be dimensioned in the subroutine.

```
SUBROUTINE SUB( MAT Q,MAT P,C )
```

Syntax

```
BEGIN CASE
    CASE expression
        statements
    [ CASE expression
        statements
            .
            .
            .          ]
END CASE
```

Description

Use the CASE statement to alter the sequence of instruction execution based on the value of one or more expressions. If *expression* in the first CASE statement is true, the following statements up to the next CASE statement are executed. Execution continues with the statement following the END CASE statement.

If the expression in a CASE statement is false, execution continues by testing the expression in the next CASE statement. If it is true, the statements following the CASE statement up to the next CASE or END CASE statement are executed. Execution continues with the statement following the END CASE statement.

If more than one CASE statement contains a true expression, only the statements following the first such CASE statement are executed. If no CASE statements are true, none of the statements between the BEGIN CASE and END CASE statements are executed.

If an expression evaluates to the null value, the CASE statement is considered false.

Use the ISNULL function with the CASE statement when you want to test whether the value of a variable is the null value. This is the only way to test for the null value since null cannot be equal to any value, including itself. The syntax is:

CASE ISNULL (*expression*)

Use an expression of the constant "1" to specify a default CASE to be executed if none of the other CASE expressions evaluate to true.

Examples

In the following example NUMBER is equal to 3. CASE 1 is always true, therefore control is transferred to subroutine 30. Once the subroutine RETURN is executed, control proceeds to the statement following the END CASE statement.

```
NUMBER=3
BEGIN CASE
   CASE NUMBER=1
      GOTO 10
   CASE 1
      GOSUB 30
   CASE NUMBER<3
      GOSUB 20
END CASE
PRINT 'STATEMENT FOLLOWING END CASE'
GOTO 50
10*
PRINT 'LABEL 10'
STOP
20*
PRINT 'LABEL 20'
RETURN
30*
PRINT 'LABEL 30'
RETURN
50*
```

This is the program output:

```
LABEL 30
STATEMENT FOLLOWING END CASE
```

In the following example, control proceeds to the statement following the END CASE because 'NAME' does not meet any of the conditions:

```
NAME="MICHAEL"
BEGIN CASE
   CASE NAME[1,2]='DA'
      PRINT NAME
      GOTO 10
   CASE NAME[1,2]='RI'
      PRINT NAME
      GOSUB 20
```

```
    CASE NAME[1,2]='BA'
       PRINT NAME
       GOSUB 30
END CASE
PRINT 'NO MATCH'
STOP
```

This is the program output:

```
NO MATCH
```

CATS function

Syntax

CATS (*array1*, *array2*)

CALL –CATS (*return.array*, *array1*, *array2*)

CALL !CATS (*return.array*, *array1*, *array2*)

Description

Use the CATS function to create a dynamic array of the element-by-element concatenation of two dynamic arrays.

Each element of *array1* is concatenated with the corresponding element of *array2*. The result is returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, the existing element is returned. If an element of one dynamic array is the null value, null is returned for the concatenation of the corresponding elements.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A="A":@VM:"B":@SM:"C"
B="D":@SM:"E":@VM:"F"
PRINT CATS(A,B)
```

This is the program output:

```
ADsEvBFsC
```

Syntax

CHAIN *command*

Description

Use the CHAIN statement to terminate execution of a BASIC program and to execute the value of *command*. *command* is an expression that evaluates to any valid UniVerse command. If *command* evaluates to the null value, the CHAIN statement fails, the program terminates, and a run-time error message is produced.

Local variables belonging to the current program are lost when you chain from one program to another. Named and unnamed common variables are retained.

CHAIN differs from the EXECUTE or PERFORM statements in that CHAIN does not return control to the calling program. If a program chains to a proc, any nested calling procs are removed.

PICK, IN2, and REALITY Flavors

Unnamed common variables are lost when a chained program is invoked in a PICK, IN2, or REALITY flavor account. If you want to save the values of variables in unnamed common, use the KEEP.COMMON keyword to the RUN command at execution.

Example

The following program clears the screen, initializes the common area, and then runs the main application:

```
PRINT @(-1)
PRINT "INITIALIZING COMMON, PLEASE WAIT"
GOSUB INIT.COMMON
CHAIN "RUN BP APP.MAIN KEEP.COMMON"
```

CHANGE function

Syntax

CHANGE (*expression*, *substring*, *replacement* [,*occurrence* [,*begin*]])

Description

Use the CHANGE function to replace a substring in *expression* with another substring. Each occurrence of the substring is replaced.

occurrence specifies the number of occurrences of *substring* to replace. To change all occurrences, specify *occurrence* as a number less than 1.

begin specifies the first occurrence to replace. If *begin* is less than 1, it defaults to 1.

If *substring* is an empty string, the value of *expression* is returned. If *replacement* is an empty string, all occurrences of *substring* are removed.

If *expression* evaluates to the null value, null is returned. If *substring*, *replacement*, *occurrence*, or *begin* evaluate to the null value, the CHANGE function fails, the program terminates, and a run-time error message is produced.

Example

```
A = "AAABBBCCCDDDBBB"
PRINT CHANGE (A,"BBB","ZZZ")
PRINT CHANGE (A,"","ZZZ")
PRINT CHANGE (A,"BBB","")
```

This is the program output:

```
AAAZZZCCCDDDZZZ
AAABBBCCCDDDBBB
AAACCCDDD
```

Syntax

CHAR (*expression*)

Description

Use the CHAR function to generate an ASCII character from the numeric value of *expression*.

If *expression* evaluates to the null value, null is returned. If *expression* evaluates to 128, CHAR(128) is returned, not the null value. CHAR(128) is the equivalent of the system variable @NULL.STR.

The CHAR function is the inverse of the SEQ function.

Example

```
X = CHAR(38)
Y = CHAR(32)
PRINT X:Y:X
```

CHAR(38) is an ampersand (&). CHAR(32) is a space. This is the program output:

```
& &
```

CHARS function

Syntax

CHARS (*dynamic.array*)

CALL –CHARS (*return.array, dynamic.array*)

CALL !CHARS (*return.array, dynamic.array*)

Description

Use the CHARS function to generate a dynamic array of ASCII characters from the decimal numeric value of each element of *dynamic.array.*

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

If any element in the dynamic array is the null value, null is returned for that element. If any element in the dynamic array evaluates to 128, CHAR(128) is returned, not the null value. CHAR(128) is the equivalent of the system variable @NULL.STR.

Example

```
X = CHARS(38:@VM:32:@VM:38)
PRINT X
```

The dynamic array X comprises three elements: CHAR(38) (an ampersand (&)), CHAR(32) (a space), and another CHAR(38). The program prints a dynamic array of these elements separated by value marks:

```
&v v&
```

Syntax

CHECKSUM (*string*)

Description

Use the CHECKSUM function to return a cyclical redundancy code (a checksum value).

If *string* is the null value, null is returned.

Example

```
A = "THIS IS A RECORD TO BE SENT VIA SOME PROTOCOL"
REC = A:@FM:CHECKSUM(A)
PRINT REC
```

This is the program output:

```
THIS IS A RECORD TO BE SENT VIA SOME PROTOCOLF30949
```

CLEAR statement

Syntax

CLEAR [COMMON]

Description

Use the CLEAR statement at the beginning of a program to set all assigned and unassigned values of variables outside of the common area of the program to 0. This procedure avoids run-time errors for unassigned variables. If you use the CLEAR statement later in the program, any values assigned to noncommon variables (including arrays) are lost.

Use the COMMON option to reset the values of all the variables in the unnamed common area to 0. Variables outside the common area or in the named common area are unaffected.

Example

```
A=100
PRINT "The value of A before the CLEAR statement:"
PRINT A
CLEAR
PRINT "The value of A after the CLEAR statement:"
PRINT A
PRINT
*
COMMON B,C,D
D="HI"
PRINT "The values of B, C, and D"
PRINT B,C,D
CLEAR COMMON
PRINT B,C,D
```

This is the program output:

```
The value of A before the CLEAR statement: 100
The value of A after the CLEAR statement:    0
The values of B, C, and D
0        0         HI
0        0         0
```

Syntax

CLEARDATA

Description

Use the CLEARDATA statement to flush all data that has been loaded in the input stack by the DATA statement. No expressions or spaces are allowed with this statement. Use the CLEARDATA statement when an error is detected, to prevent data placed in the input stack from being used incorrectly.

Example

The following program is invoked from a paragraph. A list of filenames and record IDs is passed to it from the paragraph via DATA statements. If a file cannot be opened, the CLEARDATA statement clears the data stack since the DATA statements would no longer be valid to the program.

```
TEN:
INPUT FILENAME
IF FILENAME="END" THEN STOP
OPEN FILENAME TO FILE ELSE
   PRINT "CAN'T OPEN FILE ":FILENAME
   PRINT "PLEASE ENTER NEW FILENAME "
   CLEARDATA
   GOTO TEN:
END
TWENTY:
INPUT RECORD
READ REC FROM FILE,RECORD ELSE GOTO TEN:
PRINT REC<1>
GOTO TEN:
```

```
TEST.FILE.
0 records listed.
```

CLEARFILE statement

Syntax

CLEARFILE [*file.variable*] [ON ERROR *statements*] [LOCKED *statements*]

Description

Use the CLEARFILE statement to delete all records in an open dictionary or data file. You cannot use this statement to delete the file itself. Each file to be cleared must be specified in a separate CLEARFILE statement.

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement).

The CLEARFILE statement fails, the program terminates, and a run-time error message is produced if:

- The file is neither accessible nor open.
- *file.variable* evaluates to the null value.
- A distributed file contains a part file that cannot be accessed, but the CLEARFILE statement clears those part files still available.
- A transaction is active. That is, you cannot execute this statement between a BEGIN TRANSACTION (or TRANSACTION START) statement and the COMMIT (or TRANSACTION START) or ROLLBACK statement that ends the transaction.

The ON ERROR Clause

The ON ERROR clause is optional in the CLEARFILE statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the CLEARFILE statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message displays.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number. If a CLEARFILE statement is used when any portion of a file is locked, the program waits until the file is released. The ON ERROR clause is not supported if the CLEARFILE statement is within a transaction.

The LOCKED Clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the CLEARFILE statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file
- Intent file
- Shared file
- Update record
- Shared record

If the CLEARFILE statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

Example

```
OPEN "","TEST.FILE" ELSE PRINT "NOT OPEN"
EXECUTE "LIST TEST.FILE"
CLEARFILE
CHAIN "LIST TEST.FILE"
```

This is the program output:

```
LIST TEST.FILE 11:37:45am  03-22-94 PAGE    1
TEST.FILE
ONE
TWO
THREE
3 records listed.
LIST TEST.FILE 11:37:46am  03-22-94 PAGE    1
TEST.FILE.
0 records listed.
```

CLEARPROMPTS statement

Syntax

CLEARPROMPTS

CALL !CLEAR.PROMPTS

Description

Use the CLEARPROMPTS statement to clear the value of the in-line prompt. Once a value is entered for an in-line prompt, the prompt continues to have that value until a CLEARPROMPTS statement is executed, unless the in-line prompt control option A is specified. CLEARPROMPTS clears all values that have been entered for in-line prompts.

For information about in-line prompts, see the ILPROMPT function.

Syntax

CLEARSELECT [ALL | *list.number*]

Description

Use the CLEARSELECT statement to clear an active select list. This statement is normally used when one or more select lists have been generated but are no longer needed. Clearing select lists prevents remaining select list entries from being used erroneously.

Use the keyword ALL to clear all active select lists. Use *list.number* to specify a numbered select list to clear. *list.number* must be a numeric value from 0 through 10. If neither ALL nor *list.number* is specified, select list 0 is cleared.

If *list.number* evaluates to the null value, the CLEARSELECT statement fails, the program terminates, and a run-time error message is produced.

PICK, REALITY, and IN2 Flavors

PICK, REALITY, and IN2 flavor accounts store select lists in list variables instead of numbered select lists. In those accounts, and in programs that use the VAR.SELECT option of the $OPTIONS statement, the syntax of CLEARSELECT is:

CLEARSELECT [ALL | *list.variable*]

Example

The following program illustrates the use of CLEARSELECT to clear a partially used select list. The report is designed to display the first 40-odd hours of lessons. A CLEARSELECT is used so that all the selected records are not printed. Once the select list is cleared, the READNEXT ELSE clause is executed.

```
OPEN 'SUN.SPORT' TO FILE ELSE STOP "CAN'T OPEN FILE"
HOURS=0
*
EXECUTE 'SSELECT SUN.SPORT BY START BY INSTRUCTOR'
*
START:
READNEXT KEY ELSE
   PRINT 'FIRST WEEK', HOURS
   STOP
END
```

```
READ MEMBER FROM FILE,KEY ELSE GOTO START:
HOURS=HOURS+MEMBER<4>

PRINT MEMBER<1>,MEMBER<4>
IF HOURS>40 THEN
   ******
   CLEARSELECT
   ******
   GOTO START:
END
GOTO START:
END
```

This is the program output:

```
14 records selected to Select List #0
4309     1
6100     4
3452     3
6783     12
5390     9
4439     4
6203     14
FIRST WEEK         47
```

Syntax

CLOSE [*file.variable*] [ON ERROR *statements*]

Description

Use the CLOSE statement after opening and processing a file. Any file locks or record locks are released.

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed. If the file is neither accessible nor open, or if *file.variable* evaluates to the null value, the CLOSE statement fails, the program terminates, and a run-time error message is produced.

The ON ERROR Clause

The ON ERROR clause is optional in the CLOSE statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the CLOSE statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

Example

```
CLEAR
OPEN '','EX.BASIC' TO DATA ELSE STOP
READ A FROM DATA, 'XYZ' ELSE STOP
A<3>='*'
WRITE A ON DATA, 'XYZ'
CLOSE DATA
```

Syntax

CLOSESEQ *file.variable* [ON ERROR *statements*]

Description

Use the CLOSESEQ statement after opening and processing a file opened for sequential processing. CLOSESEQ makes the file available to other users.

file.variable specifies a file previously opened with an OPENSEQ statement. If the file is neither accessible nor open, the program terminates and a run-time error message is produced. If *file.variable* is the null value, the CLOSESEQ statement fails, the program terminates, and a run-time error message is produced.

The ON ERROR Clause

The ON ERROR clause is optional in the CLOSESEQ statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the CLOSESEQ statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

Example

In this example, the CLOSESEQ statement closes FILE.E, making it available to other users:

```
OPENSEQ 'FILE.E', 'RECORD1' TO FILE ELSE ABORT
READSEQ A FROM FILE THEN PRINT A ELSE STOP
CLOSESEQ FILE
END
```

Syntax

COL1 ( )

Description

Use the COL1 function after the execution of a FIELD function to return the numeric value for the character position that immediately precedes the selected substring (see the FIELD function). Although the COL1 function takes no arguments, parentheses are required to identify it as a function.

The value obtained from COL1 is local to the program or subroutine executing the FIELD function. Before entering a subroutine, the current value of COL1 in the main program is saved. The value of COL1 in the subroutine is initialized as 0. When control is returned to the calling program, the saved value of COL1 is restored.

If no FIELD function precedes the COL1 function, a value of 0 is returned. If the delimiter expression of the FIELD function is an empty string or the null value, or if the string is not found, the COL1 function returns a 0 value.

Examples

The FIELD function in the following example returns the substring CCC. COL1( ) returns 8, the position of the delimiter ($) that precedes CCC.

```
SUBSTRING=FIELD("AAA$BBB$CCC",'$',3)
POS=COL1()
```

In the following example, the FIELD function returns a substring of 2 fields with the delimiter (**.**) that separates them: 4.5. COL1( ) returns 6, the position of the delimiter that precedes 4.

```
SUBSTRING=FIELD("1.2.3.4.5",'.',4,2)
POS=COL1()
```

COL2 function

Syntax

COL2 ( )

Description

Use the COL2 function after the execution of a FIELD function to return the numeric value for the character position that immediately follows the selected substring (see the FIELD function). Although the COL2 function takes no arguments, parentheses are required to identify it as a function.

The value obtained from COL2 is local to the program or subroutine executing the FIELD function. Before entering a subroutine, the current value of COL2 in the main program is saved. The value of COL2 in the subroutine is initialized as 0. When control is returned to the calling program, the saved value of COL2 is restored.

If no FIELD function precedes the COL2 function, a value of 0 is returned. If the delimiter expression of the FIELD function is an empty string or the null value, or if the string is not found, the COL2 function returns a 0 value.

Examples

The FIELD function in the following example returns the substring 111. COL2( ) returns 4, the position of the delimiter (#) that follows 111.

```
SUBSTRING=FIELD("111#222#3","#",1)
P=COL2()
```

In the following example, the FIELD function returns a substring of two fields with the delimiter (&) that separates them: 7&8. COL2( ) returns 5, the position of the delimiter that follows 8.

```
SUBSTRING=FIELD("&7&8&B&","&",2,2)
S=COL2()
```

In the next example, FIELD( ) returns the whole string, because the delimiter (**.**) is not found. COL2( ) returns 6, the position after the last character of the string.

```
SUBSTRING=FIELD("9*8*7",".",1)
Y=COL2()
```

In the next example, FIELD( ) returns an empty string, because there is no tenth occurrence of the substring in the string. COL2( ) returns 0 because the substring was not found.

```
SUBSTRING=FIELD("9*8*7","*",10)
O=COL2()
```

COMMIT statement

Syntax

COMMIT [WORK] [THEN *statements*] [ELSE *statements*]

Description

Use the COMMIT statement to commit all file I/O changes made during a transaction. The WORK keyword is provided for compatibility with SQL syntax conventions; it is ignored by the compiler.

A transaction includes all statements between a BEGIN TRANSACTION statement and the COMMIT or ROLLBACK statement that ends the transaction. Either a COMMIT or a ROLLBACK statement ends the current transaction.

The COMMIT statement can either succeed or fail.

When a subtransaction commits, it makes the results of its database operations accessible to its parent transaction. The subtransaction commits to the database only if all of its predecessors up to the top-level transaction are committed.

If a top-level transaction succeeds, all changes to files made during the active transaction are committed to disk.

If a subtransaction fails, all its changes are rolled back and do not affect the parent transaction. If the top-level transaction fails, none of the changes made during the active transaction are committed, and the database remains unaffected by the failed transaction. This ensures that the database is maintained in a consistent state.

If the COMMIT statement succeeds, the THEN statements are executed; any ELSE statements are ignored. If COMMIT fails, any ELSE statements are executed. After the THEN or the ELSE statements are executed, control is transferred to the statement following the next END TRANSACTION statement.

All locks obtained during a transaction remain in effect for the duration of the active transaction; they are not released by a RELEASE, WRITE, WRITEV, or MATWRITE statement that is part of the transaction. The parent transaction adopts the acquired or promoted locks. If a subtransaction rolls back, any locks that have been acquired or promoted within that transaction are demoted or released.

The COMMIT statement that ends the top-level transaction releases locks set during that transaction. Locks obtained outside the transaction are not affected by the COMMIT statement.

If no transaction is active, the COMMIT statement generates a run-time warning, and the ELSE statements are executed.

Example

This example begins a transaction that applies locks to rec1 and rec2. If no errors occur, the COMMIT statement ensures that the changes to rec1 and rec2 are written to the file. The locks on rec1 and rec2 are released, and control is transferred to the statement following the END TRANSACTION statement.

```
BEGIN TRANSACTION
   READU data1 FROM file1,rec1 ELSE ROLLBACK

   READU data2 FROM file2,rec2, ELSE ROLLBACK
       .
       .
       .
   WRITE new.data1 ON file1,rec1 ELSE ROLLBACK
   WRITE new.data2 ON file2,rec2 ELSE ROLLBACK
   COMMIT WORK
END TRANSACTION
```

Note that the update record lock on rec1 is not released on completion of the first WRITE statement but on completion of the COMMIT statement.

COMMON statement

Syntax

COM[MON] [/ *name* /] *variable* [,*variable* ...]

Description

Use the COMMON statement to provide a storage area for variables. Variables in the common area are accessible to main programs and external subroutines. Corresponding variables can have different names in the main program and in external subroutines, but they must be defined in the same order. The COMMON statement must precede any reference to the variables it names.

A common area can be either named or unnamed. An unnamed common area is lost when the program completes its execution and control returns to the UniVerse command level. A named common area remains available for as long as the user remains in the UniVerse environment.

The common area name can be of any length, but only the first 31 characters are significant.

Arrays can be dimensioned and named with a COMMON statement. They can be redimensioned later with a DIMENSION statement, but the COMMON statement must appear before the DIMENSION statement. When an array is dimensioned in a subroutine, it takes on the dimensions of the array in the main program regardless of the dimensions stated in the COMMON statement. For a description of dimensioning array variables in a subroutine, see the CALL statement.

When programs share a common area, use the $INCLUDE statement to define the common area in each program.

Example

Program:

```
COMMON NAME, ADDRESS (15, 6), PHONE
```

Subroutine:

```
COMMON A, B (15, 6), C
```

In this example the variable pairs NAME and A, ADDRESS and B, PHONE and C are stored in the same memory location.

Syntax

COMPARE (*string1*, *string2* [,*justification*])

Description

Use the COMPARE function to compare two strings and return a numeric value indicating the result.

string1, *string2* specify the strings to be compared.

justification is either L for left-justified comparison or R for right-justified comparison. (Any other value causes a run-time warning, and 0 is returned.)

The comparison can be left-justified or right-justified. A right-justified comparison compares numeric substrings within the specified strings as numbers. The numeric strings must occur at the same character position in each string. For example, a right-justified comparison of the strings AB100 and AB99 indicates that AB100 is greater than AB99 since 100 is greater than 99. A right-justified comparison of the strings AC99 and AB100 indicates that AC99 is greater since C is greater than B.

If neither L nor R is specified, the default comparison is left-justified.

The following list shows the values returned:

–1	*string1* is less than *string2*.
0	*string1* equals *string2* or the justification expression is not valid.
1	*string1* is greater than *string2*.

Example

In the following example, the strings AB99 and AB100 are compared with the right-justified option and the result displayed. In this case the result displayed is –1.

```
PRINT COMPARE('AB99','AB100','R')
```

CONTINUE statement

The CONTINUE statement is a loop-controlling statement. For syntax details, see the FOR statement and the LOOP statement.

Syntax

CONVERT (*expression1*, *expression2*, *variable*)

Description

Use the CONVERT function to return a copy of *variable* with every occurrence of specified characters in *variable* replaced with other specified characters. Every time a character to be converted appears in *variable*, it is replaced by the replacement character.

expression1 specifies a list of characters to be converted. *expression2* specifies the corresponding replacement characters. The first character of *expression2* replaces all instances of the first character of *expression1*, the second character of *expression2* replaces all instances of the second character of *expression1*, and so on.

If *expression2* contains more characters than *expression1*, the extra characters are ignored. If *expression1* contains more characters than *expression2*, the characters with no corresponding *expression2* characters are deleted from the result.

If *variable* is the null value, null is returned. If either *expression1* or *expression2* is the null value, the CONVERT function fails, the program terminates, and a run-time error message is produced.

The CONVERT function works similarly to the CONVERT statement.

Example

```
A="NOW IS THE TIME"
PRINT A
A=CONVERT('TI','XY',A)
PRINT A
A=CONVERT('XY','T',A)
PRINT A
```

This is the program output:

```
NOW IS THE TIME
NOW YS XHE XYME
NOW S THE TME
```

CONVERT statement

Syntax

CONVERT *expression1* TO *expression2* IN *variable*

Description

Use the CONVERT statement to replace every occurrence of specific characters in a string with other characters. Every time the character to be converted appears in the string, it is replaced by the replacement character.

expression1 specifies a list of characters to be converted. *expression2* specifies a list of replacement characters. The first character of *expression2* replaces all instances of the first character of *expression1*, the second character of *expression2* replaces all instances of the second character of *expression1*, and so on.

If *expression2* contains more characters than *expression1*, the extra characters are ignored. If *expression1* contains more characters than *expression2*, the characters with no corresponding *expression2* characters are deleted from the variable.

If *variable* is the null value, null is returned. If either *expression1* or *expression2* evaluates to the null value, the CONVERT statement fails, the program terminates, and a run-time error message is produced.

Example

```
A="NOW IS THE TIME"
PRINT A
CONVERT 'TI' TO 'XY' IN A
PRINT A
CONVERT 'XY' TO 'T' IN A
PRINT A
```

This is the program output:

```
NOW IS THE TIME
NOW YS XHE XYME
NOW S THE TME
```

Syntax

COS (*expression*)

Description

Use the COS function to return the trigonometric cosine of an angle. *expression* is an angle expressed as a numeric value in degrees. The COS function is the inverse of the ACOS function.

Values outside the range of 0 to 360 degrees are interpreted as modulo 360. Numbers greater than 1E17 produce a warning message and 0 is returned. If *expression* evaluates to the null value, null is returned.

Example

```
PRINT "COS(45) = " : COS(45)
END
```

This is the program output:

```
COS(45) = 0.7071
```

COSH function

Syntax

COSH (*expression*)

Description

Use the COSH function to return the hyperbolic cosine of *expression*. *expression* must be a numeric value.

If *expression* evaluates to the null value, null is returned.

Example

```
PRINT "COSH(2) = ":COSH(2)
```

This is the program output:

```
COSH(2) = 3.7622
```

Syntax

COUNT (*string*, *substring*)

Description

Use the COUNT function to return the number of times a substring is repeated in a string value.

string is an expression that evaluates to the string value to be searched. *substring* is an expression that evaluates to the substring to be counted. *substring* can be a character string, a constant, or a variable.

If *substring* does not appear in *string*, a 0 value is returned. If *substring* is an empty string, the number of characters in *string* is returned. If *string* is the null value, null is returned. If *substring* is the null value, the COUNT function fails, the program terminates, and a run-time error message is produced.

By default, each character in string is matched to *substring* only once. Therefore, when *substring* is longer than one character and a match is found, the search continues with the character following the matched substring. No part of the matched string is recounted toward another match. For example, the following statement counts two occurrences of substring TT and assigns the value 2 to variable C:

```
C = COUNT ('TTTT', 'TT')
```

PICK, IN2, and REALITY Flavors

In PICK, IN2, and REALITY flavors, the COUNT function continues the search with the next character regardless of whether it is part of the matched string. For example, the following statement counts three occurrences of substring TT:

```
C = COUNT ('TTTT', 'TT')
```

Use the COUNT.OVLP option of the $OPTIONS statement to get this behavior in IDEAL and INFORMATION flavor accounts.

Example

```
A=COUNT('ABCAGHDALL','A')
PRINT "A= ",A
*
Z='S#FF##G#JJJJ#'
```

```
Q=COUNT(Z,'#')
PRINT "Q= ",Q
*
Y=COUNT('11111111','11')
PRINT "Y= ",Y
```

This is the program output:

```
A=       3
Q=       5
Y=       4
```

Syntax

COUNTS (*dynamic.array, substring*)

CALL –COUNTS (*return.array, dynamic.array, substring*)

CALL !COUNTS (*return.array, dynamic.array, substring*)

Description

Use the COUNTS function to count the number of times a substring is repeated in each element of a dynamic array. The result is a new dynamic array whose elements are the counts corresponding to the elements in *dynamic.array.*

dynamic.array specifies the dynamic array whose elements are to be searched.

substring is an expression that evaluates to the substring to be counted. *substring* can be a character string, a constant, or a variable.

Each character in an element is matched to *substring* only once. Therefore, when *substring* is longer than one character and a match is found, the search continues with the character following the matched substring. No part of the matched element is recounted toward another match.

If *substring* does not appear in an element, a 0 value is returned. If *substring* is an empty string, the number of characters in the element is returned. If *substring* is the null value, the COUNTS function fails, the program terminates, and a run-time error message is produced.

If any element in *dynamic.array* is the null value, null is returned.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

PICK, IN2, and REALITY Flavors

In PICK, IN2, and REALITY flavors, the COUNTS function continues the search with the next character regardless of whether it is part of the matched string. Use the COUNT.OVLP option of the $OPTIONS statement to get this behavior in IDEAL and INFORMATION flavor accounts.

Example

```
ARRAY="A":@VM:"AA":@SM:"AAAAA"
PRINT COUNTS(ARRAY, "A")
PRINT COUNTS(ARRAY, "AA")
```

This is the program output:

```
1v2s5
0v1s2
```

Syntax

CREATE *file.variable* {THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the CREATE statement after an OPENSEQ statement to create a record in a type 1 or type 19 UniVerse file or to create a UNIX or DOS file. CREATE creates the record or file if the OPENSEQ statement fails. An OPENSEQ statement for the specified *file.variable* must be executed before the CREATE statement to associate the pathname or record ID of the file to be created with the *file.variable*. If *file.variable* is the null value, the CREATE statement fails, the program terminates, and a run-time error message is produced.

Use the CREATE statement when OPENSEQ cannot find a record or file to open and the next operation is to be a NOBUF, READSEQ, or READBLK. You need not use the CREATE statement if the first file operation is a WRITESEQ, since WRITESEQ creates the record or file if it does not exist.

If the record or file is created, the THEN statements are executed, and the ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next statement.

If the record or file is not created, the ELSE statements are executed; any THEN statements are ignored.

File Buffering

Normally UniVerse uses buffering for sequential input and output operations. Use the NOBUF statement after an OPENSEQ statement to turn off buffering and cause all writes to the file to be performed immediately. For more information about file buffering, see the NOBUF statement.

Example

In the following example, RECORD4 does not yet exist. When OPENSEQ fails to open RECORD4 to the file variable FILE, the CREATE statement creates RECORD4 in the type 1 file FILE.E and opens it to the file variable FILE.

```
OPENSEQ 'FILE.E', 'RECORD4' TO FILE
   ELSE CREATE FILE ELSE ABORT
WEOFSEQ FILE
WRITESEQ 'HELLO, UNIVERSE' TO FILE ELSE STOP
```

CRT statement

Syntax

CRT [*print.list*]

Description

Use the CRT statement to print data on the screen, regardless of whether a PRINTER ON statement has been executed. The syntax for *print.list* is the same as for PRINT statements.

print.list can contain any BASIC expression. The elements of the list can be numeric or character strings, variables, constants, or literal strings; the null value, however, cannot be output. The list can consist of a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no *print.list* is designated, a blank line is output.

Expressions separated by commas are printed at preset tab positions. You can use multiple commas together to cause multiple tabulation between expressions.

Expressions separated by colons are concatenated. That is, the expression following the colon is printed immediately after the expression preceding the colon. To print a list without a LINEFEED and RETURN, end the *print.list* with a colon (:).

The CRT statement works similarly to the DISPLAY statement.

Example

```
CRT "This can be used to print something on the"
CRT "terminal while"
CRT "the PRINTER ON statement is in effect."
```

The program output on the terminal is:

```
This can be used to print something on the
terminal while
the PRINTER ON statement is in effect.
```

Syntax

DATA *expression* [,*expression* ...]

Description

Use the DATA statement to place values in an input stack. These values can be used as responses to INPUT statements executed later in the program or in a subroutine (see the INPUT statement). The values can also serve as responses to UniVerse commands that request input.

Expressions used in DATA statements can be numeric or character string data. The null value cannot be stored in the input stack. If *expression* evaluates to null, the DATA statement fails, the program terminates, and a run-time error message is produced.

Put a comma at the end of each line of a DATA statement to indicate that more data expressions follow on the next line.

The order in which expressions are specified in the DATA statement is the order in which the values are accessed by subsequent INPUT statements: first-in, first-out. When all DATA values have been exhausted, the INPUT statement prompts the user for a response at the terminal.

The DATA statement must be executed before an INPUT statement that is to use *expression* for input.

You can store up to 512 characters in a data stack.

You can list the current data in the stack from your program by accessing the @DATA.PENDING variable with the statement:

```
PRINT @DATA.PENDING
```

Example

In the following example, the INPUT NBR statement uses the first value placed in the input stack by the DATA statement, 33, as the value of NBR. The INPUT DESCR statement uses the second value, 50, as the value of DESCR. The INPUT PRICE statement uses the third value, 21, as the value of PRICE.

```
X=33;  Y=50; Z=21
DATA X,Y,Z
X=Y+Z
```

```
*
INPUT NBR
INPUT DESCR
INPUT PRICE
INPUT QTY
PRINT NBR,DESCR,PRICE,QTY
```

This is the program output:

```
?33
?50
?21
?2
33       50        21        2
```

Note that the value of NBR is the value of X when the DATA statement is executed, not the current value of X (namely, Y+Z). The INPUT QTY statement has no corresponding value in the input stack, so it prompts the user for input.

Syntax

DATE ( )S

Description

Use the DATE function to return the numeric value of the internal system date. Although the DATE function takes no arguments, parentheses are required to identify it as a function.

The internal format for the date is based on a reference date of December 31, 1967, which is day 0. All dates thereafter are positive numbers representing the number of days elapsed since day 0. All dates before day 0 are negative numbers representing the number of days before day 0. For example:

Internal Formats for Dates

Date	Internal Representation
December 10, 1967	–21
November 15, 1967	–46
February 15, 1968	46
January 1, 1985	6575

Example

```
PRINT DATE()
PRINT OCONV(DATE(),"D2/")
```

This is the program output:

```
9116
12/15/92
```

DCOUNT function

Syntax

DCOUNT (*string*, *delimiter*)

Description

Use the DCOUNT function to return the number of delimited fields in a data string.

string is an expression that evaluates to the data string to be searched.

delimiter is an expression that evaluates to the delimiter separating the fields to be counted. *delimiter* can be a character string of 0, 1, or more characters.

DCOUNT differs from COUNT in that it returns the number of values separated by delimiters rather than the number of occurrences of a character string. Two consecutive delimiters in *string* are counted as one field. If *delimiter* evaluates to an empty string, a count of 1 plus the number of characters in the string is returned. If *string* evaluates to an empty string, 0 is returned.

If *string* evaluates to the null value, null is returned. If *delimiter* evaluates to the null value, the DCOUNT function fails, the program terminates, and a run-time error message is produced.

PICK, IN2, and REALITY Flavors

In PICK, IN2, and REALITY flavors, the DCOUNT function continues the search with the next character regardless of whether it is part of the matched delimiter string. Use the COUNT.OVLP option of the $OPTIONS statement to get this behavior in IDEAL and INFORMATION flavor accounts.

Example

```
REC="88.9.B.7"
Q=DCOUNT(REC,'.')
PRINT "Q= ",Q
REC=34:@VM:55:@VM:88:@VM:"FF":@VM:99:@VM:"PP"
R=DCOUNT(REC,@VM)
PRINT "R= ",R
```

This is the program output:

```
Q=        4
R=        6
```

Syntax

DEBUG

Description

Use the DEBUG statement to invoke RAID, the interactive UniVerse BASIC debugger. The DEBUG statement takes no arguments. When this statement is encountered, program execution stops and the double colon (:) prompt appears, waiting for a RAID command. The following table summarizes the RAID commands:

RAID Commands

Command	Action
line	Displays the specified line of the source code.
/[*string*]	Searches the source code for *string*.
B	Set a RAID breakpoint.
C	Continue program execution.
D	Delete a RAID breakpoint.
G	Go to a specified line or address and continue program execution.
H	Display statistics for the program.
I	Display and execute the next object code instruction.
L	Print the next line to be executed.
M	Set watchpoints.
Q	Quit RAID.
R	Run the program.
S	Step through the BASIC source code.
T	Display the call stack trace.
V	Enter verbose mode for the M command.
V*	Print the compiler version that generated the object code.
W	Display the current window.
X	Display the current object code instruction and address.

RAID Commands (Continued)

Command	Action
X*	Display local run machine registers and variables.
Z	Display the next 10 lines of source code.
$	Turn on instruction counting.
#	Turn on program timing.
+	Increment the current line or address.
–	Decrement the current line or address.
.	Display the last object code instruction executed.
variable /	Print the value of *variable*.
variable!*string*	Change the value of *variable* to *string*.

Syntax

DEFFUN *function* [([MAT] *argument* [, [MAT] *argument* ...])]
[CALLING *call.ID*]

Description

Use the DEFFUN statement to define a user-written function. You must declare a user-defined function before you can use it in a program. The DEFFUN statement provides the compiler with information such as the function name and the number and type of arguments. You can define a user-written function only once in a program. A subsequent DEFFUN statement for an already defined user-written function causes a fatal error.

function is the name of the user-written function.

arguments supply up to 254 arguments in the DEFFUN statement. To pass an array, you must precede the array name with the keyword MAT. An extra argument is hidden so that the user-defined function can use it to return a value. An extra argument is retained by the user-written function so that a value is returned by a RETURN (*value*) statement (for more information see the RETURN (*value*) statement). If the RETURN (*value*) statement specifies no value, an empty string is returned. The extra argument is reported by the MAP and MAKE.MAP.FILE commands.

call.ID is an expression that evaluates to the name by which the function is called if it is not the same as the function name. It can be a quoted string (the call ID itself) or a variable that evaluates to the call ID. If you do not use the CALLING clause, the user-defined function is presumed to be defined in the VOC file and cataloged without any prefix.

Examples

The following example defines a user-written function called MYFUNC with the arguments or formal parameters A, B, and C:

```
FUNCTION MYFUNC(A, B, C)
Z = ...
RETURN (Z)
END
```

The next example declares the function MYFUNC. It uses the function with the statement T = MYFUNC (X, Y, Z). The actual parameters held in X, Y, and Z are referenced by the formal parameters A, B, and C, so the value assigned to T can be calculated.

```
DEFFUN MYFUNC(X, Y, Z)
T = MYFUNC(X, Y, Z)
END
```

Syntax

DEL *dynamic.array* < *field#* [,*value#* [,*subvalue#*]] >

Description

Use the DEL statement to delete a field, value, or subvalue from a dynamic array. The DEL statement works similarly to the DELETE function.

dynamic.array is an expression that evaluates to a dynamic array. If *dynamic.array* evaluates to the null value, null is returned.

field# is an expression that evaluates to the field in *dynamic.array. value#* is an expression that evaluates to the value in the field. *subvalue#* is an expression that evaluates to the subvalue in the value. These expressions are called delimiter expressions. The numeric values of the delimiter expressions specify which field, value, or subvalue to delete. The entire position is deleted, including its delimiter characters.

value# and *subvalue#* are optional. If they are equal to 0, the entire field is deleted. If *subvalue#* is equal to 0 and *value#* and *field#* are greater than 0, the specified value in the specified field is deleted. If all three delimiter expressions are greater than 0, only the specified subvalue is deleted.

If any delimiter expression is the null value, the DEL statement fails, the program terminates, and a run-time error message is produced.

If a higher-level delimiter expression has a value of 0 when a lower-level delimiter expression is greater than 0, the 0 delimiter is treated as if it were equal to 1. The delimiter expressions are, from highest to lowest: field, value, and subvalue.

If the DEL statement references a subelement of a higher element whose value is the null value, the dynamic array is unchanged. Similarly, if all delimiter expressions are 0, the original string is returned.

Examples

In the following examples a field mark is shown by **F**, a value mark is shown by **V**, and a subvalue mark is shown by **S**.

The first example deletes field 1 and sets Q to VAL1vSUBV1sSUBV2FFSUBV3sSUBV4:

```
R="FLD1":@FM:"VAL1":@VM:"SUBV1":@SM:"SUBV2":@FM:@FM:"SUBV3":@SM:"SUBV4"
Q=R
DEL Q<1,0,0>
```

The next example deletes the first subvalue in field 4 and sets the value of Q to FLD1FVAL1vSUBV1sSUBV2FFSUBV4:

```
Q=R
DEL Q<4,1,1>
```

The next example deletes the second value in field 2 and sets the value of Q to FLD1FVAL1FFSUBV3sSUBV4:

```
Q=R
DEL Q<2,2,0>
```

The next example deletes field 3 entirely and sets the value of Q to FLD1FVAL1vSUBV1sSUBV2FSUBV3sSUBV4:

```
Q=R
DEL Q<3,0,0>
```

The next example deletes the second subvalue in field 4 and sets the value of Q to FLD1FVAL1vSUBV1sSUBV2FFSUBV3:

```
Q=R
DEL Q<4,1,2>
```

Syntax

DELETE (*dynamic.array, field#*[,*value#*[,*subvalue#*]])

Description

Use the DELETE function to erase the data contents of a specified field, value, or subvalue and its corresponding delimiter from a dynamic array. The DELETE function returns the contents of the dynamic array with the specified data removed without changing the actual value of the dynamic array.

dynamic.array is an expression that evaluates to the array in which the field, value, or subvalue to be deleted can be found. If *dynamic.array* evaluates to the null value, null is returned.

field# is an expression that evaluates to the field in the dynamic array; *value#* is an expression that evaluates to the value in the field; *subvalue#* is an expression that evaluates to the subvalue in the value. The numeric values of the delimiter expressions specify which field, value, or subvalue to delete. The entire position is deleted, including its delimiting characters.

value# and *subvalue#* are optional. If they are equal to 0, the entire field is deleted. If *subvalue#* is equal to 0 and *value#* and *field#* are greater than 0, the specified value in the specified field is deleted. If all three delimiter expressions are greater than 0, only the specified subvalue is deleted.

If any delimiter expression is the null value, the DELETE function fails, the program terminates, and a run-time error message is produced.

If a higher-level delimiter expression has a value of 0 when a lower-level delimiter is greater than 0, the 0 delimiter is treated as if it were equal to 1. The delimiter expressions are, from highest to lowest: field, value, and subvalue.

If the DELETE function references a subelement of a higher element whose value is the null value, the dynamic array is unchanged. Similarly, if all delimiter expressions are 0, the original string is returned.

Examples

In the following examples a field mark is shown by **F**, a value mark is shown by **V**, and a subvalue mark is shown by **S**.

The first example deletes field 1 and sets Q to VAL1vSUBV1sSUBV2FFSUBV3sSUBV4:

```
R="FLD1":@FM:"VAL1":@VM:"SUBV1":@SM:"SUBV2":@FM:@FM:"SUBV3":@SM:"SUBV4"
Q=DELETE (R,1)
```

The next example deletes the first subvalue in field 4 and sets the value of Q to FLD1FVAL1vSUBV1sSUBV2FFSUBV4:

```
Q=DELETE (R,4,1,1)
```

The next example deletes the second value in field 2 and sets the value of Q to FLD1FVAL1FFSUBV3sSUBV4:

```
Q=DELETE (R,2,2)
```

The next example deletes field 3 entirely and sets the value of Q to FLD1FVAL1vSUBV1sSUBV2FSUBV3sSUBV4:

```
Q=DELETE (R,3,0,0)
```

The next example deletes the second subvalue in field 4 and sets the value of Q to FLD1FVAL1vSUBV1sSUBV2FFSUBV3:

```
Q=DELETE (R,4,1,2)
```

Syntax

DELETE [*file.variable*,] *record.ID* [ON ERROR *statements*]
[LOCKED *statements*]
[THEN *statements*] [ELSE *statements*]

DELETEU [*file.variable*,] *record.ID* [ON ERROR *statements*]
[LOCKED *statements*]
[THEN *statements*] [ELSE *statements*]

Description

Use the DELETE statements to delete a record from a UniVerse file. If you specify a file variable, the file must be open when the DELETE statement is encountered (see the OPEN statement).

file.variable is a file variable from a previous OPEN statement.

record.ID is an expression that evaluates to the record ID of the record to be deleted.

If the file does not exist or is not open, the program terminates and a run-time error results. If you do not specify a file variable, the most recently opened default file is used (see the OPEN statement for more information on default files). If you specify both a file variable and a record ID, you must use a comma to separate them.

If the file is an SQL table, the effective user of the program must have SQL DELETE privilege to delete records in the file. For information about the effective user of a program, see the AUTHORIZATION statement.

The record is deleted, and any THEN statements are executed. If the deletion fails, the ELSE statements are executed; any THEN statements are ignored.

If a record is locked, it is not deleted, and an error message is produced. The ELSE statements are not executed.

If either *file.variable* or *record.ID* evaluates to the null value, the DELETE statement fails, the program terminates, and a run-time error message is produced.

The DELETEU Statement

Use the DELETEU statement to delete a record without releasing the update record lock set by a previous READU statement (see the READ statements in Volume 2).

The file must have been previously opened with an OPEN statement. If a file variable was specified in the OPEN statement, it can be used in the DELETEU statement. You must place a comma between the file variable and the record ID expression. If no file variable is specified in the DELETEU statement, the statement applies to the default file. See the OPEN statement in Volume 2 for a description of the default file.

The ON ERROR Clause

The ON ERROR clause is optional in the DELETE statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the DELETE statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED Clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the DELETE statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file
- Intent file

- Shared file
- Update record
- Shared record

If the DELETE statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

Releasing the Record Lock

A record lock held by a DELETEU statement can be released explicitly with a RELEASE statement or implicitly with a WRITE, WRITEV, MATWRITE, or DELETE statement. The record lock is released when you return to the UniVerse system prompt.

Examples

```
OPEN "","MLIST" TO MALIST ELSE STOP
PRINT "FILE BEFORE DELETE STATEMENT:"
EXECUTE "COUNT MLIST"
PRINT
DELETE MALIST, "JONES"
PRINT "FILE AFTER DELETE STATMENT:"
EXECUTE "LIST MLIST"
```

This is the program output:

```
FILE BEFORE DELETE STATEMENT:

3 records listed.

FILE AFTER DELETE STATMENT:

2 records listed.
```

In the following example, the data portion of the SUBSIDIARIES files is opened to the file variable SUBS. If the file cannot be opened an appropriate message is printed. The record MADRID is read and then deleted from the file. An update record lock had been set and is maintained by the DELETEU statement.

```
OPEN "","SUBSIDIARIES" TO SUBS
READU REC FROM SUBS, 'MADRID'
   ELSE STOP 'Sorry, cannot open Subsidiaries file'
DELETEU SUBS, "MADRID"
```

DELETELIST statement

Syntax

DELETELIST *listname*

Description

Use the DELETELIST statement to delete a select list saved in the &SAVEDLISTS& file.

listname can evaluate to the form:

record.ID

or:

record.ID account.name

record.ID is the name of a select list in the &SAVEDLISTS& file. If you specify *account.name*, the &SAVEDLISTS& file of the specified account is used instead of the local &SAVEDLISTS& file.

If *listname* evaluates to the null value, the DELETELIST statement fails, the program terminates, and a run-time error message is produced.

DELETEU statement

Use the DELETEU statement to maintain an update record lock while performing the DELETE statement.

DIMENSION statement

Syntax

DIM[ENSION] *matrix* (*rows*, *columns*) [, *matrix* (*rows*, *columns*) ...]

DIM[ENSION] *vector* (*subscript*) [, *vector* (*subscript*) ...]

Description

Use the DIMENSION statement to define the dimensions of an array variable before referencing the array in the program. For a matrix (a two-dimensional array), use the DIMENSION statement to set the maximum number of rows and columns available for the elements of the array. For a vector (a one-dimensional array), use the DIMENSION statement to set the maximum value of the subscript (the maximum elements) in the array.

matrix and *vector* can be any valid variable name. The maximum dimension can be any valid numeric expression. When specifying the two dimensions of a matrix, you must use a comma to separate the row and column expressions. These expressions are called indices.

You can use a single DIMENSION statement to define multiple arrays. If you define more than one array with a DIMENSION statement, you must use commas to separate the array definitions.

The DIMENSION statement declares only the name and size of the array. It does not assign values to the elements of the array. Assignment of values to the elements is done with the MAT, MATPARSE, MATREAD, MATREADU, and assignment statements.

The DIMENSION statement in an IDEAL or INFORMATION flavor account is executed at run time. The advantage of the way UniVerse handles this statement is that the amount of memory allocated is not determined until the DIM statement is executed. This means that arrays can be redimensioned at run time.

When redimensioning an array, you can change the maximum number of elements, rows, columns, or any combination thereof. You can even change the dimensionality of an array (that is, from a one-dimensional to a two-dimensional array or vice versa).

The values of the array elements are affected by redimensioning as follows:

- Common elements (those with the same indices) are preserved.

- New elements (those that were not indexed in the original array) are initialized as unassigned.
- Abandoned elements (those that can no longer be referenced in the altered array) are lost, and the memory space is returned to the operating system.

The DIMENSION statement fails if there is not enough memory available for the array. When this happens, the INMAT function is set to a value of 1.

An array variable that is passed to a subroutine in its entirety as an argument in a CALL statement cannot be redimensioned in the subroutine. Each array in a subroutine must be dimensioned once. The dimensions declared in the subroutine DIMENSION statement are ignored, however, when an array is passed to the subroutine as an argument (for more information, see the CALL statement).

PICK, IN2, and REALITY Flavors

In PICK, IN2, and REALITY flavor accounts, arrays are created at compile time, not run time. Arrays are not redimensionable, and they do not have a zero element. To get the same characteristics in an INFORMATION or IDEAL flavor account, use the STATIC.DIM option of the $OPTIONS statement.

Examples

```
DIM ARRAY(2,2)
ARRAY(1,1)="KK"
ARRAY(1,2)="GG"
ARRAY(2,1)="MM"
ARRAY(2,2)="NN"
```

In the next example warning messages are printed for the unassigned elements in the matrix. The elements are assigned empty strings as their values.

```
DIM ARRAY(2,3)
*
PRINT
FOR X=1 TO 2
   FOR Y=1 TO 3
      PRINT "ARRAY(":X:",":Y:")", ARRAY(X,Y)
   NEXT Y
NEXT X
DIM S(3,2)
S(1,1)=1
S(1,2)=2
```

```
S(2,1)=3
S(2,2)=4
S(3,1)=5
S(3,2)=6
```

In the next example the common elements are preserved. Those elements that cannot be referenced in the new matrix (S(3,1), S(3,2)) are lost.

```
DIM S(2,2)
*
PRINT
FOR X=1 TO 2
FOR Y=1 TO 2
PRINT "S(":X:",":Y:")", S(X,Y)
NEXT Y
NEXT X
```

This is the program output:

```
ARRAY(1,1)         KK
ARRAY(1,2)         GG
ARRAY(1,3)         Program 'DYNAMIC.DIMENSION':
Line 12, Variable previously undefined, empty string used.

ARRAY(2,1)         MM
ARRAY(2,2)         NN
ARRAY(2,3)         Program 'DYNAMIC.DIMENSION':
Line 12, Variable previously undefined, empty string used.

S(1,1)   1
S(1,2)   2
S(2,1)   3
S(2,2)   4
```

Syntax

DISPLAY [*print.list*]

Description

Use the DISPLAY statement to print data on the screen, regardless of whether a PRINTER ON statement has been executed. The syntax for *print.list* is the same as for PRINT statements.

The elements of the list can be numeric or character strings, variables, constants, or literal strings; the null value, however, cannot be output. The list can consist of a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no *print.list* is designated, a blank line is output.

Expressions separated by commas are printed at preset tab positions. You can use multiple commas together to cause multiple tabulation between expressions.

Expressions separated by colons are concatenated. That is, the expression following the colon is printed immediately after the expression preceding the colon. To print a list without a LINEFEED and RETURN, end the print list with a colon (:).

The DISPLAY statement works similarly to the CRT statement.

Example

```
DISPLAY "This can be used to print something on the
DISPLAY "terminal while"
DISPLAY "the PRINTER ON statement is in effect."
```

The program output on the terminal is:

```
This can be used to print something on the
terminal while
the PRINTER ON statement is in effect.
```

DIV function

Syntax

DIV (*dividend*, *divisor*)

Description

Use the DIV function to calculate the value of the quotient after *dividend* is divided by *divisor*.

The dividend and divisor expressions can evaluate to any numeric value. The only exception is that *divisor* cannot be 0. If either *dividend* or *divisor* evaluates to the null value, null is returned.

Example

```
X=100;  Y=25
Z = DIV (X,Y)
PRINT Z
```

This is the program output:

```
4
```

Syntax

DIVS (*array1*, *array2*)

CALL –DIVS (*return.array*, *array1*, *array2*)

CALL !DIVS (*return.array*, *array1*, *array2*)

Description

Use the DIVS function to create a dynamic array containing the result of the element-by-element division of two dynamic arrays.

Each element of *array1* is divided by the corresponding element of *array2* with the result being returned in the corresponding element of a new dynamic array. If elements of *array1* have no corresponding elements in *array2*, *array2* is padded with ones and the *array1* elements are returned. If an element of *array2* has no corresponding element in *array1*, 0 is returned. If an element of *array2* is 0, a run-time error message is printed and a 0 is returned. If either element of a corresponding pair is the null value, null is returned.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A=10:@VM:15:@VM:9:@SM:4
B=2:@VM:5:@VM:9:@VM:2
PRINT DIVS(A,B)
```

This is the program output:

```
5v3v1s4v0
```

DOWNCASE function

Syntax

DOWNCASE (*expression*)

Description

Use the DOWNCASE function to change all uppercase letters in *expression* to lowercase. If *expression* evaluates to the null value, null is returned.

Example

```
A="DOWN CASE DOES THIS:  "
PRINT A:DOWNCASE(A)
B="Down Case Does This:  "
PRINT B:DOWNCASE(B)
```

This is the program output:

```
DOWN CASE DOES THIS:  down case does this:
Down Case Does This:  down case does this:
```

Syntax

DQUOTE (*expression*)

Description

Use the DQUOTE function to enclose an expression in double quotation marks. If *expression* evaluates to the null value, null is returned.

Example

```
PRINT DQUOTE(12 + 5) : " IS THE ANSWER."
END
```

This is the program output:

```
"17" IS THE ANSWER.
```

DTX function

Syntax

DTX (*number* [,*size*])

Description

Use the DTX function to convert a decimal integer to its hexadecimal equivalent.

size indicates the minimum size which the hexadecimal character string should have. This field is supplemented with zeros if appropriate.

If *number* evaluates to the null value, null is returned. If *size* is the null value, the DTX function fails, the program terminates, and a run-time error message is produced.

Example

```
X = 25
Y = DTX (X)
PRINT Y
Y = DTX (X,4)
PRINT Y
END
```

This is the program output:

```
19
0019
```

Syntax

EBCDIC (*expression*)

Description

Use the EBCDIC function to convert each character of *expression* from its ASCII representation value to its EBCDIC representation value. The EBCDIC and ASCII functions perform complementary operations. Data that is not represented in ASCII code produces undefined results.

If *expression* evaluates to the null value, the EBCDIC function fails, the program terminates, and a run-time error message is produced.

Example

```
X = 'ABC 123'
Y = EBCDIC(X)
PRINT "ASCII", "EBCDIC", " X "
PRINT "------", "-----", "---"
FOR I = 1 TO LEN (X)
PRINT SEQ(X[I,1]) , SEQ(Y[I,1]),X[I,1]
NEXT I
```

This is the program output:

```
ASCII     EBCDIC   X
------    -----    ---
65        193      A
66        194      B
67        195      C
32        64
49        241      1
50        242      2
51        243      3
```

ECHO statement

Syntax

ECHO {ON | OFF | *expression*}

Description

Use the ECHO statement to control the display of input characters on the screen.

If ECHO ON is specified, subsequent input characters are displayed, or echoed, on the screen. If ECHO OFF is specified, subsequent input characters are assigned to the INPUT statement variables but are not displayed on the screen.

The ability to turn off character display is useful when the keyboard is to be used for cursor movement or for entering password information. If *expression* evaluates to true, ECHO is turned ON. If *expression* evaluates to false, ECHO is turned OFF. If *expression* evaluates to the null value, it is treated as false, and ECHO is turned OFF.

Example

```
PROMPT ""
ECHO OFF
PRINT "ENTER YOUR PASSWORD"
INPUT PWORD
ECHO ON
```

This is the program output:

```
ENTER YOUR PASSWORD
```

Syntax

END

Description

Use the END statement to terminate a BASIC program or a section of an IF, READ, or OPEN statement.

An END statement is the last statement in a UniVerse BASIC program; it indicates the logical end of the program. When an END statement that is not associated with an IF, READ, or OPEN statement is encountered, execution of the program terminates. You can use comments after the END statement.

You can also use the END statement with conditional statements in the body of a program. In this case END indicates the end of a multistatement conditional clause.

INFORMATION and REALITY Flavors

In INFORMATION and REALITY flavors a warning message is printed if there is no final END statement. The END.WARN option of the $OPTIONS statement prints the warning message in IDEAL, IN2, PICK, and PIOPEN flavors under the same conditions.

Example

```
A="YES"
IF A="YES" THEN
   PRINT "THESE TWO LINES WILL PRINT ONLY"
   PRINT "WHEN THE VALUE OF 'A' IS 'YES'."
END
*
PRINT
PRINT "THIS IS THE END OF THE PROGRAM"
END ; * END IS THE LAST STATEMENT EXECUTED
```

This is the program output:

```
THESE TWO LINES WILL PRINT ONLY
WHEN THE VALUE OF 'A' IS 'YES'.

THIS IS THE END OF THE PROGRAM
```

END CASE statement

Use the END CASE statement to end a set of CASE statements.

END TRANSACTION statement

Use the END TRANSACTION statement to specify where processing is to continue after a transaction ends.

ENTER statement

Syntax

ENTER *subroutine*

variable = '*subroutine*'
ENTER @*variable*

Description

Use the ENTER statement to transfer program control from the calling program to an external subroutine without returning to the calling program. The subroutine must have been compiled and cataloged.

The ENTER statement is similar to the CALL statement, except that with the ENTER statement, program flow does not return from the entered program to the calling program (see the CALL statement). The ENTER statement also does not accept arguments.

In the PIOPEN flavor, the ENTER statement is a synonym for the CALL statement. It takes arguments and returns control to the calling program.

External subroutines can be entered directly or indirectly. To enter a subroutine indirectly, assign the name of the cataloged subroutine to a variable or to an element of an array. Use the name of this variable or array element, prefixed with an at sign (@), as the operand of the ENTER statement.

If *subroutine* evaluates to the null value, the ENTER statement fails, the program terminates, and a run-time error message is produced.

Example

The following program transfers control to the cataloged program PROGRAM2:

```
ENTER PROGRAM2
```

EOF(ARG.) function

Syntax

EOF(ARG.)

Description

Use the EOF(ARG.) function to check if the command line argument pointer is past the last command line argument. ARG. is part of the syntax of the EOF(ARG.) function and must be specified. EOF(ARG.) returns 1 (true) if the pointer is past the last command line argument, otherwise it returns 0 (false).

The *arg#* argument of the GET(ARG.) and SEEK(ARG.) statements affect the value of the EOF(ARG.) function.

EQS function

Syntax

EQS (*array1*, *array2*)

CALL –EQS (*return.array*, *array1*, *array2*)

CALL !EQS (*return.array*, *array1*, *array2*)

Description

Use the EQS function to test if elements of 1 dynamic array are equal to the elements of another dynamic array.

Each element of *array1* is compared with the corresponding element of *array2*. If the two elements are equal, a 1 is returned in the corresponding element of a dynamic array. If the two elements are not equal, a 0 is returned. If an element of 1 dynamic array has no corresponding element in the other dynamic array, a 0 is returned. If either element of a corresponding pair is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array returns as *return.array*.

Example

```
A=1:@VM:45:@SM:3:@VM:"one"
B=0:@VM:45:@VM:1
PRINT EQS(A,B)
```

This is the program output:

```
0v1s0v0
```

Syntax

EQU[ATE] *symbol* TO *expression* [,*symbol* TO *expression* ...]

EQU[ATE] *symbol* LIT[ERALLY] *string* [,*symbol* LIT *string* ...]

Description

In an EQUATE statement, *symbol* represents the value of *expression* or *string*. You can use the two interchangeably in the program. When the program is compiled, each occurrence of *symbol* is replaced by the value of *expression* or *string*. The value is compiled as object code and does not have to be reassigned each time the program is executed.

You can define multiple symbols in a single EQUATE statement. *symbol* cannot be a number.

You can define *symbol* only once. Any subsequent EQUATE state generates a compiler error because the compiler interprets the symbol before the statement is parsed.

If you use TO as a connector, the object can be any BASIC expression. If you use LIT or LITERALLY as a connector, the object must be a literal string.

RAID does not recognize EQUATE symbols. You must use the object value in RAID sessions.

There is no limit on the number of EQUATE statements allowed by the BASIC compiler, except that of memory.

If *symbol* is the same as the name of a BASIC function, the function is disabled in the program. If a statement exists with the same name as a disabled function, the statement is also disabled.

Examples

In the following example, A is made equivalent to the string JANE:

```
JANE="HI"
EQUATE A TO "JANE"
```

Next, B is made equivalent to the variable JANE:

```
EQUATE B LIT "JANE"
PRINT "A IS EQUAL TO ":A
PRINT "B IS EQUAL TO ":B
```

This is the program output:

```
A IS EQUAL TO JANE
B IS EQUAL TO HI
```

In the next example COST is made equivalent to the value of the expression PRICE*QUANTITY:

```
EQUATE COST LIT "PRICE * QUANTITY"
PRICE=3;QUANTITY=7
PRINT "THE TOTAL COST IS $": COST
```

This is the program output:

```
THE TOTAL COST IS $21
```

The next example shows an EQUATE statement with multiple symbols:

```
EQUATE C TO "5",
   D TO "7",
   E LIT "IF C=5 THEN PRINT 'YES'"
PRINT "C+D=": C+D
E
```

This is the program output:

```
C+D=12
YES
```

Syntax

EREPLACE (*expression*, *substring*, *replacement* [,*occurrence* [,*begin*]])

Description

Use the EREPLACE function to replace *substring* in *expression* with another substring. If you do not specify *occurrence*, each occurrence of *substring* is replaced.

occurrence specifies the number of occurrences of *substring* to replace. To replace all occurrences, specify *occurrence* as a number less than 1.

begin specifies the first occurrence to replace. If *begin* is less than 1, it defaults to 1.

If *substring* is an empty string, *replacement* is prefixed to *expression*. If *replacement* is an empty string, all occurrences of *substring* are removed.

Example

```
A = "AAABBBCCCDDDBBB"
PRINT EREPLACE (A,"BBB","ZZZ")
PRINT EREPLACE (A,"","ZZZ")
PRINT EREPLACE (A,"BBB","")
```

This is the program output:

```
AAAZZZCCCDDDZZZ
ZZZAAABBBCCCDDDBBB
AAACCCDDD
```

ERRMSG statement

Syntax

ERRMSG *message.ID* [,*message.ID* ...]

Description

Use the ERRMSG statement to print a formatted error message from the ERRMSG file.

message.ID is an expression evaluating to the record ID of a message to be printed on the screen. Additional expressions are evaluated as arguments that can be included in the error message.

If *message.ID* evaluates to the null value, the default error message is printed:

```
Message ID is NULL:  undefined error
```

A standard Pick ERRMSG file is supplied with UniVerse. Users can construct a local ERRMSG file using the following syntax in the records. Each field must start with one of these codes shown in the following table:

ERRMSG File Codes

Code	Action
A[(*n*)]	Display next argument left-justified; *n* specifies field length.
D	Display system date.
E [*string*]	Display record ID of message in brackets; *string* displayed after ID.
H [*string*]	Display *string*.
L [(*n*)]	Output a newline; *n* specifies number of newlines.
R [(*n*)]	Display next argument right-justified; *n* specifies field length.
S [(*n*)]	Output *n* blank spaces from beginning of line.
T	Display system time.

Example

```
>ED ERRMSG 1
7 lines long.
----: P
0001: HBEGINNING OF ERROR MESSAGE
0002: L
```

```
0003: HFILE NAMED "
0004: A

0005: H" NOT FOUND.
0006: L
0007: H END OF MESSAGE
Bottom at line 7
----: Q
OPEN 'SUN.SPORT' TO test
THEN PRINT "File Opened" ELSE ERRMSG "1", "SUN.SPORT"
```

This is the program output:

```
BEGINNING OF ERROR MESSAGE
FILE NAMED "SUN.SPORT" NOT FOUND.
END OF MESSAGE
```

EXCHANGE function

Syntax

EXCHANGE (*string*, *xx*, *yy*)

Description

Use the EXCHANGE function to replace one character with another or to delete all occurrences of the specified character.

string is an expression evaluating to the string whose characters are to be replaced or deleted. If *string* evaluates to the null value, null is returned.

xx is an expression evaluating to the character to be replaced, in hexadecimal notation.

yy is an expression evaluating to the replacement character, also in hexadecimal notation.

If *yy* is FF, all occurrences of *xx* are deleted. If *xx* or *yy* consist of fewer than two characters, no conversion is done. If *xx* or *yy* is the null value, the EXCHANGE function fails, the program terminates, and a run-time error message is produced.

Note: 0x80 is treated as @NULL.STR, not as @NULL.

Example

In the following example, 41 is the hexadecimal value for the character A and 2E is the hexadecimal value for the period character (`.`):

```
PRINT EXCHANGE('ABABC','41','2E')
```

This is the program output:

```
.B.BC
```

Syntax

EXECUTE *commands* [CAPTURING *variable*] [PASSLIST [*dynamic.array*]]
[RTNLIST [*variable*]] [{SETTING | RETURNING} *variable*]

EXECUTE *commands* [,IN < *expression*] [,OUT > *variable*]
[,SELECT[(*list*)] < *dynamic.array*] [,SELECT[(*list*)] > *variable*]
[,PASSLIST [(*dynamic.array*)]] [,STATUS > *variable*]

EXECUTE *commands* [, / / IN. < *expression*] [, / / OUT. > *variable*]
[, / / SELECT.[(*list*)] < *dynamic.array*] [, / / SELECT.[(*list*)] > *variable*]
[, / / PASSLIST.[(*dynamic.array*)]] [, / / STATUS. > *variable*]

Description

Use the EXECUTE statement to execute UniVerse commands from within the BASIC program and then return execution to the statement following the EXECUTE statement.

EXECUTE creates a new environment for the executed command. This new environment is initialized with the values of the current prompt, current printer state, **Break** key counter, the values of in-line prompts, KEYEDITs, KEYTRAPs, and KEYEXITs. If any of these values change in the new environment, the changes are not passed back to the calling environment. In the new environment, stacked @variables are either initialized to 0 or set to reflect the new environment. Nonstacked @variables are shared between the EXECUTE and calling environments.

commands can be sentences, paragraphs, verbs, procs, menus, or BASIC programs. You can specify multiple commands in the EXECUTE statement in the same way they are specified in a UniVerse paragraph. Each command or line must be separated by a field mark (ASCII CHAR 254).

The EXECUTE statement has two main syntaxes. The first syntax requires options to be separated by spaces. The second and third syntaxes require options to be separated by commas. In the third syntax, the "/ /" preceding the keywords and the periods following them are optional; the compiler ignores these marks. Except for the slashes and periods, the second and third syntaxes are the same.

In the first syntax the CAPTURING clause assigns the output of the executed commands to *variable*. The PASSLIST clause passes the current active select list or expression to the commands for use as select list 0. The RTNLIST option assigns select list 0, created by the commands, to *variable*. If you do not specify *variable*, the RTNLIST clause is ignored. Using the SETTING or RETURNING clause causes the @SYSTEM.RETURN.CODE of the last executed command to be placed in *variable*.

In the second syntax the executed commands use the value of *expression* in the IN clause as input. When the IN clause is used, the DATA queue is passed back to the calling program, otherwise data is shared between environments. The OUT clause assigns the output of the commands to *variable*. The SELECT clauses let you supply the select list stored in *expression* as a select list to the commands, or to assign a select list created by the commands to *variable*. If *list* is not specified, select list 0 is used. The PASSLIST clause passes the currently active select list to the commands. If you do not specify *list*, select list 0 in the current program's environment is passed as select list 0 in the executed command's environment. The STATUS clause puts the @SYSTEM.RETURN.CODE of the last executed command in *variable*.

The EXECUTE statement fails, the program terminates, and a run-time error message is produced if:

- *dynamic.array* or *expression* in the IN clause evaluates to the null value.
- The command expression evaluates to the null value.

Although you cannot use the EXECUTE statement within a transaction to execute most UniVerse commands and SQL statements, you can use EXECUTE to execute the following UniVerse commands and SQL statements within a transaction:

UniVerse Commands and SQL Statements Used with EXECUTE

CHECK.SUM	INSERT	SEARCH	SSELECT
COUNT	LIST	SELECT (RetrieVe)	STAT
DELETE (SQL)	LIST.ITEM	SELECT (SQL)	SUM
DISPLAY	LIST.LABEL	SORT	UPDATE
ESEARCH	RUN	SORT.ITEM	
GET.LIST	SAVE.LIST	SORT.LABEL	

INFORMATION Flavor

In INFORMATION flavor accounts, the EXECUTE statement without any options is the same as the PERFORM statement. In this case executed commands keep the same environment as the BASIC program that called them. Use the EXEC.EQ.PERF option of the $OPTIONS statement to cause EXECUTE to behave like PERFORM in other flavors.

Example

The following example performs a nested SELECT, demonstrating the use of the CAPTURING, RTNLIST, and PASSLIST keywords:

```
CMD = "SELECT VOC WITH TYPE = V"
EXECUTE CMD RTNLIST VERBLIST1
CMD = "SELECT VOC WITH NAME LIKE ...LIST..."
EXECUTE CMD PASSLIST VERBLIST1 RTNLIST VERBLIST2
CMD = "LIST VOC NAME"
EXECUTE CMD CAPTURING RERUN PASSLIST VERBLIST2
PRINT RERUN
```

The program first selects all VOC entries that define verbs, passing the select list to the variable VERBLIST1. Next, it selects from VERBLIST1 all verbs whose names contain the string LIST and passes the new select list to VERBLIST2. The list in VERBLIST2 is passed to the LIST command, whose output is captured in the variable RERUN, which is then printed.

EXIT statement

Syntax

EXIT

Description

Use the EXIT statement to quit execution of a LOOP...REPEAT loop and branch to the statement following the REPEAT statement of the loop. The EXIT statement quits exactly one loop. When loops are nested and the EXIT statement is executed in an inner loop, the outer loop remains in control.

Example

```
COUNT = 0
LOOP
WHILE COUNT < 100 DO
   INNER = 0
   LOOP
   WHILE INNER < 100 DO
      COUNT += 1
      INNER += 1
      IF INNER = 50 THEN EXIT
   REPEAT
   PRINT "COUNT = ":COUNT
REPEAT
```

This is the program output:

```
COUNT = 50
COUNT = 100
```

Syntax

EXP (*expression*)

Description

Use the EXP function to return the value of "e" raised to the power designated by *expression*. The value of "e" is approximately 2.71828. *expression* must evaluate to a numeric value.

If *expression* is too large or small, a warning message is printed and 0 is returned. If *expression* evaluates to the null value, null is returned.

The formula used by the EXP function to perform the calculations is

value of EXP function = 2.71828**(*expression*)

Example

```
X=5
PRINT EXP(X-1)
```

This is the program output:

```
54.5982
```

EXTRACT function

Syntax

EXTRACT (*dynamic.array, field#* [,*value#* [,*subvalue#*]])

variable < *field#* [,*value#* [,*subvalue#*]] >

Description

Use the EXTRACT function to access the data contents of a specified field, value, or subvalue from a dynamic array. You can use either syntax shown to extract data. The first syntax uses the EXTRACT keyword, the second uses angle brackets.

dynamic.array is an expression that evaluates to the array in which the field, value, or subvalue to be extracted is to be found. If *dynamic.array* evaluates to the null value, null is returned.

field# specifies the field in the dynamic array; *value#* specifies the value in the field; *subvalue#* specifies the subvalue in the value. These arguments are called delimiter expressions. The numeric values of the delimiter expressions determine whether a field, a value, or a subvalue is to be extracted. *value#* and *subvalue#* are optional.

Angle brackets used as an EXTRACT function appear on the right side of an assignment statement. Angle brackets on the left side of the assignment statement indicate that a REPLACE function is to be performed (for examples, see the REPLACE function in Volume 2).

The second syntax uses angle brackets to extract data from dynamic arrays. *variable* specifies the dynamic array containing the data to be extracted. *field#*, *value#*, and *subvalue#* are delimiter expressions.

There are five possible outcomes that can result from the different uses of delimiter expressions; they are described as cases 1 through 5.

Case 1: If *field#*, *value#*, and *subvalue#* are omitted or evaluate to 0, an empty string is returned.

Case 2: If *value#* and *subvalue#* are omitted or evaluate to 0, the entire field is extracted.

Case 3: If *subvalue#* is omitted or specified as 0 and *value#* and *field#* evaluate to nonzero, the entire specified value in the specified field is extracted.

Case 4: If *field#*, *value#*, and *subvalue#* are all specified and are all nonzero, the specified subvalue is extracted.

Case 5: If *field#*, *value#*, or *subvalue#* evaluates to the null value, the EXTRACT function fails, the program terminates, and a run-time error message is produced.

If a higher-level delimiter expression has a value of 0 when a lower-level delimiter is greater than 0, a 1 is assumed. The delimiter expressions are from highest to lowest: field, value, and subvalue.

If the EXTRACT function references a subelement of an element whose value is the null value, null is returned.

Example

In the following example a field mark is shown by **F**, a value mark is shown by **V**, and a subvalue mark is shown by **S**:

```
VAR=1:@FM:4:@VM:9:@SM:3:@SM:5:@FM:1:@VM:0:@SM:7:@SM:3
Z=EXTRACT(VAR,1,0,0)
PRINT Z
*
Z=VAR<1,1,1>
PRINT Z
*
Z=EXTRACT(VAR,2,1,1)
PRINT Z
*
Z=VAR<3,2,3>
PRINT Z
*
Z=EXTRACT(VAR,10,0,0)
PRINT Z
*
Z=EXTRACT(VAR,2,2,0)
PRINT Z
*
```

This is the program output:

```
1
1
4
3
9s3s5
```

FADD function

Syntax

FADD (*number1*, *number2*)

CALL !FADD (*return.array*, *number1*, *number2*)

Description

Use the FADD function to perform floating-point addition on two numeric values. If either number evaluates to the null value, null is returned. If either *number1* or *number2* evaluates to the null value, null is returned. *return.array* equates to *number1* plus *number2*.

This function is provided for compatibility with existing software. You can also use the + operator to perform floating-point addition.

Example

```
PRINT FADD(.234,.567)
```

This is the program output:

```
0.801
```

Syntax

FDIV (*number1*, *number2*)

CALL !FDIV (*return.array*, *number1*, *number2*)

Description

Use the FDIV function to perform floating-point division on two numeric values. *number1* is divided by *number2*. *return.array* equates to *number1* divided by *number2*. If *number2* is 0, a run-time error message is produced and a 0 is returned for the function. If either number evaluates to the null value, null is returned.

This function is provided for compatibility with existing software. You can also use the / operator to perform floating-point division.

Example

```
PRINT FDIV(.234,.567)
```

This is the program output:

```
0.4127
```

FFIX function

Syntax

FFIX (*number*)

Description

Use the FFIX function to convert a floating-point number to a numeric string with fixed precision. If *number* evaluates to the null value, null is returned.

This function is provided for compatibility with existing software.

FFLT function

Syntax

FFLT (*number*)

Description

Use the FFLT function to round a number to a string with a precision of 13. The number also converts to scientific notation when required for precision. If *number* evaluates to the null value, null is returned.

FIELD function

Syntax

FIELD (*string*, *delimiter*, *occurrence* [,*num.substr*])

Description

Use the FIELD function to return 1 or more substrings located between specified delimiters in *string*.

delimiter evaluates to any character, including field mark, value mark, and subvalue marks. It delimits the start and end of the substring. If *delimiter* evaluates to more than one character, only the first character is used. Delimiters are not returned with the substring.

occurrence specifies which occurrence of the delimiter is to be used as a terminator. If *occurrence* is less than 1, 1 is assumed.

num.substr specifies the number of delimited substrings to return. If the value of *num.substr* is an empty string or less than 1, 1 is assumed. When more than one substring is returned, delimiters are returned along with the successive substrings.

If either *delimiter* or *occurrence* is not in the string, an empty string is returned, unless *occurrence* specifies 1. If *occurrence* is 1 and *delimiter* is not found, the entire string is returned. If *delimiter* is an empty string, the entire string is returned.

If *string* evaluates to the null value, null is returned. If *string* contains CHAR(128) (that is, @NULL.STR), it is treated like any other character in a string. If *delimiter*, *occurrence*, or *num.substr* evaluate to the null value, the FIELD function fails, the program terminates, and a run-time error message is produced.

The FIELD function works identically to the GROUP function.

Examples

```
D=FIELD("###DHHH#KK","#",4)
PRINT "D= ",D
```

The variable D is set to DHHH because the data between the third and fourth occurrence of the delimiter # is DHHH.

```
REC="ACADABA"
E=FIELD(REC,"A",2)
PRINT "E= ",E
```

The variable E is set to "C".

```
VAR="?"
Z=FIELD("A.1234$$$$&&",VAR,3)
PRINT "Z= ",Z
```

Z is set to an empty string since "?" does not appear in the string.

```
Q=FIELD("+1+2+3ABAC","+",2,2)
PRINT "Q= ",Q
```

Q is set to "1+2" since two successive fields were specified to be returned after the second occurrence of "+".

This is the program output:

```
D=        DHHH
E=        C
Z=
Q=        1+2
```

FIELDS function

Syntax

FIELDS (*dynamic.array*, *delimiter*, *occurrence* [,*num.substr*])

CALL –FIELDS (*return.array*, *dynamic.array*, *delimiter*, *occurrence* [,*num.substr*])

CALL !FIELDS (*return.array*, *dynamic.array*, *delimiter*, *occurrence* [,*num.substr*])

Description

Use the FIELDS function to return a dynamic array of substrings located between specified delimiters in each element of *dynamic.array*.

delimiter evaluates to any character, including value and subvalue characters. It marks the start and end of the substring. If *delimiter* evaluates to more than one character, the first character is used.

occurrence specifies which occurrence of the delimiter is to be used as a terminator. If *occurrence* is less than 1, 1 is assumed.

num.substr specifies the number of delimited substrings to return. If the value of *num.substr* is an empty string or less than 1, 1 is assumed. In this case delimiters are returned along with the successive substrings. If *delimiter* or *occurrence* does not exist in the string, an empty string is returned, unless *occurrence* specifies 1. If *occurrence* is 1 and the specified delimiter is not found, the entire element is returned. If *occurrence* is 1 and *delimiter* is an empty string, an empty string is returned.

If *dynamic.array* is the null value, null is returned. If any element in *dynamic.array* is the null value, null is returned for that element. If *delimiter*, *occurrence*, or *num.substr* evaluates to the null value, the FIELDS function fails, the program terminates, and a run-time error message is produced.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A="000-P-0":@VM:"-H--O-":@SM:"N-I-T":@VM:"BC":@SM:"-L-"
PRINT FIELDS(A,"-",2)
```

This is the program output:

```
PvHsIvsL
```

Syntax

FIELDSTORE (*string*, *delimiter*, *start*, *n*, *new.string*)

Description

Use the FIELDSTORE function to modify character strings by inserting, deleting, or replacing fields separated by specified delimiters.

string is an expression that evaluates to the character string to be modified.

delimiter evaluates to any single ASCII character, including field, value, and subvalue marks.

start evaluates to a number specifying the starting field position. Modification begins at the field specified by *start*. If *start* is greater than the number of fields in *string*, the required number of empty fields is generated before the FIELDSTORE function is executed.

n specifies the number of fields of *new.string* to insert in *string*. *n* determines how the FIELDSTORE operation is executed. If *n* is positive, *n* fields in *string* are replaced with the first *n* fields of *new.string*. If *n* is negative, *n* fields in *string* are replaced with all the fields in *new.string*. If *n* is 0, all the fields in *new.string* are inserted in *string* before the field specified by *start*.

If *string* evaluates to the null value, null is returned. If *delimiter*, *start*, *n*, or *new.string* is null, the FIELDSTORE function fails, the program terminates, and a run-time error message is produced.

Example

```
Q='1#2#3#4#5'
*
TEST1=FIELDSTORE(Q,"#",2,2,"A#B")
PRINT "TEST1= ",TEST1
*
TEST2=FIELDSTORE(Q,"#",2,-2,"A#B")
PRINT "TEST2= ",TEST2
*
TEST3=FIELDSTORE(Q,"#",2,0,"A#B")
PRINT "TEST3= ",TEST3
*
TEST4=FIELDSTORE(Q,"#",1,4,"A#B#C#D")
```

```
PRINT "TEST4= ",TEST4
*

TEST5=FIELDSTORE(Q,"#",7,3,"A#B#C#D")
PRINT "TEST5= ",TEST5
```

This is the program output:

```
TEST1=   1#A#B#4#5
TEST2=   1#A#B#4#5
TEST3=   1#A#B#2#3#4#5
TEST4=   A#B#C#D#5
TEST5=   1#2#3#4#5##A#B#C
```

Syntax

FILEINFO (*file.variable*, *key*)

file.variable is the file variable of an open file.

key is a number that indicates the particular information required. These key numbers are described in the table "Keys and Values Supplied to the FILEINFO Function."

Description

Use the FILEINFO function to return information about the specified file's configuration, such as the specified file's parameters, its modulus and load, its operating system filename, and its VOC name. The information returned depends on the file type and the value of the key.

If the first argument is not a file variable, all keys except 0 return an empty string. A warning message is also displayed. A fatal error results if an invalid key is supplied.

Equate Names for Keys

An insert file of equate names is provided to let you use mnemonics rather than key numbers. The insert file, called FILEINFO.INS.IBAS, is located in the INCLUDE directory in the UV account directory. It is referenced in PIOPEN flavor accounts through a VOC file pointer called SYSCOM. Use the $INCLUDE statement to insert this file if you want to use equate names, as shown in the example. The following table lists the symbolic name, value, and description:

Keys and Values Supplied to the FILEINFO Function

Symbolic Name	Value	Description
FINFO$IS.FILEVAR	0	1 if *file.identifier* is a valid file identifier; 0 otherwise.
FINFO$VOCNAME	1	VOC name of the file.
FINFO$PATHNAME	2	Pathname of the file.

Keys and Values Supplied to the FILEINFO Function (Continued)

Symbolic Name	Value	Description
FINFO$TYPE	3	File type as follows: 1 Static hashed 3 Dynamic hashed 4 Type 1 5 Sequential 7 Distributed and Multivolume
FINFO$HASHALG	4	Hashing algorithm: 2 for GENERAL, 3 for SEQ.NUM.
FINFO$MODULUS	5	Current modulus.
FINFO$MINMODULUS	6	Minimum modulus.
FINFO$GROUPSIZE	7	Group size, in 1-KB units.
FINFO$LARGERECORDSIZE	8	Large record size.
FINFO$MERGELOAD	9	Merge load parameter.
FINFO$SPLITLOAD	10	Split load parameter.
FINFO$CURRENTLOAD	11	Current loading of the file (%).
FINFO$NODENAME	12	Empty string, if the file resides on the local system, otherwise the name of the node where the file resides.
FINFO$IS.AKFILE	13	1 if secondary indices exist on the file; 0 otherwise.
FINFO$CURRENTLINE	14	Current line number.
FINFO$PARTNUM	15	For a distributed file, returns list of currently open part numbers.
FINFO$STATUS	16	For a distributed file, returns list of status codes showing whether the last I/O operation succeeded or failed for each part. A value of –1 indicates the corresponding part file is not open.
FINFO$RECOVERYTYPE	17	
FINFO$RECOVERYID	18	
FINFO$IS.FIXED.MODULUS	19	

Keys and Values Supplied to the FILEINFO Function (Continued)

Symbolic Name	Value	Description
FINFO$NLSMAP	20	
FINFO$MAXOPTION	20	

Value Returned by the STATUS Function

If the function executes successfully, the value returned by STATUS( ) is 0. If the function fails to execute, STATUS returns a nonzero value. The following table lists the key, file type, and returned value for *key*:

FILEINFO Values Returned by File Type

Key	Dynamic	Directory	Distributed	Sequential
0	1 = file open 0 = file closed	1 = file open 0 = file closed	Dynamic array of codes: 1 = file open 0 = file closed	1 = file open 0 = file closed
1	VOC name	VOC name	VOC name	VOC name
2	File's pathname	Pathname of file	Dynamic array of complete pathnames in VOC record order (pathname as used in VOC for unavailable files)	File's pathname
3	3	4	7	5
4	2 = GENERAL 3 = SEQ.NUM	Empty string	Dynamic array of codes: 2 = GENERAL 3 = SEQ.NUM	Empty string
5	Current modulus	1	Dynamic array of the current modulus of each part file	
6	Minimum modulus	Empty string	Dynamic array of the minimum modulus of each part file	Empty string

FILEINFO Values Returned by File Type (Continued)

Key	Dynamic	Directory	Distributed	Sequential
7	Group size in disk records	Empty string	Dynamic array of the group size of each part file	Empty string
8	Large record size	Empty string	Dynamic array of the large record size of each part file	Empty string
9	Merge load value	Empty string	Dynamic array of the merge load % of each part file	Empty string
10	Split load value	Empty string	Dynamic array of the split load value of each part file	Empty string
11	Current load value	Empty string	Dynamic array of the current load value of each part file	Empty string
12	Local file: empty string Remote file: node name	Empty string	Local file = empty string; Remote file = dynamic array of node names	Empty string
13	1 = indices 2 = no indices	0	1 = common indices present 2 = none present	Empty string
15	Empty string	Empty string	Dynamic array of codes in VOC record order. Code is: empty string if part file not open; part number if file is open.	Empty string
16	Empty string	Empty string	Dynamic array of codes in VOC record order for each part file: 0 = I/O operation OK; –1 = part file unavail-able; >0 = error code	Empty string

Note: The first time that an I/O operation fails for a part file in a distributed file, an error code is returned for that part file by the FILEINFO function. For any subsequent I/O operations on the distributed file with the same unavailable part file, the FILEINFO function returns –1.

Examples

In the following example, the file containing the key equate names is inserted with the $INCLUDE statement. The file FILMS is opened and its file type displayed.

```
$INCLUDE SYSCOM FILEINFO.INS.IBAS
OPEN '','FILMS' TO FILMS
        ELSE STOP 'CANT OPEN FILE'
PRINT FILEINFO(FILMS,FINFO$TYPE)
```

In the next example, the file FILMS is opened and its file type displayed by specifying the numeric key:

```
OPEN '','FILMS' TO FILMS
        ELSE STOP 'CANT OPEN FILE'
PRINT FILEINFO(FILMS,3)
```

FILELOCK statement

Syntax

FILELOCK [*file.variable*] [,*lock.type*]
[ON ERROR *statements*] [LOCKED *statements*]

Description

Use the FILELOCK statement to acquire a lock on an entire file. This prevents other users from updating the file until the program releases it. A FILELOCK statement with the default lock type *lock.type* is equivalent to obtaining an update record lock on every record of the file.

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement in Volume 2). If the file is neither accessible nor open, the program terminates and a run-time error message is produced. If *file.variable* evaluates to the null value, the FILELOCK statement fails, the program terminates, and a run-time error message is produced.

lock.type can be one of the following:

- Shared file (FS)
- Intent file (IX)
- Exclusive file (FX)

The ON ERROR Clause

The ON ERROR clause is optional in the FILELOCK statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the FILELOCK statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number. If a FILELOCK statement is used when any portion of a file is locked, the program waits until the file is released.

The LOCKED Clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the FILELOCK statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

This requested lock...	Conflicts with these locks...
Intent file	Exclusive file Intent file Shared file Update record
Shared record	Exclusive file Update record
Update record	Intent file Shared file Update record Shared record

If the FILELOCK statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

Releasing Locks

A shared, intent, or exclusive file lock can be released by a FILEUNLOCK, RELEASE, or STOP statement.

Locks acquired or promoted within a transaction are not released when previous statements are processed.

Examples

```
OPEN '','SUN.MEMBER' TO DATA ELSE STOP "CAN'T OPEN FILE"
FILELOCK DATA LOCKED STOP 'FILE IS ALREADY LOCKED'
FILEUNLOCK DATA
OPEN '','SUN.MEMBER' ELSE STOP "CAN'T OPEN FILE"
FILELOCK LOCKED STOP 'FILE IS ALREADY LOCKED'
PRINT "The file is locked."
FILEUNLOCK
```

This is the program output:

```
The file is locked.
```

The following example acquires an intent file lock:

```
FILELOCK fvar, "INTENT" LOCKED
   owner = STATUS( )
   PRINT "File already locked by":owner
   STOP
END
```

Syntax

FILEUNLOCK [*file.variable*] [ON ERROR *statements*]

Description

Use the FILEUNLOCK statement to release a file lock set by the FILELOCK statement.

file.variable specifies a file previously locked with a FILELOCK statement. If *file.variable* is not specified, the default file with the FILELOCK statement is assumed (for more information on default files, see the OPEN statement in Volume 2). If *file.variable* evaluates to the null value, the FILEUNLOCK statement fails, the program terminates, and a run-time error message is produced.

The FILEUNLOCK statement releases only file locks set with the FILELOCK statement. Update record locks must be released with one of the other unlocking statements (for example, WRITE, WRITEV, and so on).

The ON ERROR Clause

The ON ERROR clause is optional in the FILEUNLOCK statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the FILEUNLOCK statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number. The ON ERROR clause is not supported if the FILEUNLOCK statement is within a transaction.

Example

In the following example, the first FILEUNLOCK statement unlocks the default file. The second FILEUNLOCK statement unlocks the file variable FILE.

```
OPEN '','SUN.MEMBER' ELSE STOP "CAN'T OPEN SUN.MEMBER"
FILELOCK
    .
    .
    .
FILEUNLOCK
OPEN 'EX.BASIC' TO FILE ELSE STOP
FILELOCK FILE
    .
    .
    .
FILEUNLOCK FILE
```

Syntax

FIND *element* IN *dynamic.array* [,*occurrence*] SETTING *fmc* [,*vmc* [,*smc*]]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the FIND statement to locate an element in *dynamic.array*. The field, value, and subvalue positions of *element* are put in the variables *fmc*, *vmc*, and *smc* respectively.

element evaluates to a character string. FIND succeeds only if the string matches an element in its entirety. If *element* is found in *dynamic.array*, any THEN statements are executed. If *element* is not found, or if *dynamic.array* evaluates to the null value, *fmc*, *vmc*, and *smc* are unchanged, and the ELSE statements are executed.

If *occurrence* is unspecified, it defaults to 1. If *occurrence* is the null value, the FIND statement fails, the program terminates, and a run-time error message is produced.

Example

```
A="THIS":@FM:"IS":@FM:"A":@FM:"DYNAMIC":@FM:"ARRAY"
FIND "IS" IN A SETTING FM,VM,SM ELSE ABORT
PRINT "FM=",FM
PRINT "VM=",VM
PRINT "SM=",SM
```

This is the program output:

```
FM=    2
VM=    1
SM=    1
```

FINDSTR statement

Syntax

FINDSTR *substring* IN *dynamic.array* [,*occurrence*]
SETTING *fmc* [,*vmc* [,*smc*]]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the FINDSTR statement to locate *substring* in *dynamic.array*. The field, value, and subvalue positions of *substring* are placed in the variables *fmc*, *vmc*, and *smc* respectively.

FINDSTR succeeds if it finds *substring* as part of any element in *dynamic array*. If *substring* is found in *dynamic.array*, any THEN statements are executed. If *substring* is not found, or if *dynamic.array* evaluates to the null value, *fmc*, *vmc*, and *smc* are unchanged, and the ELSE statements are executed.

If *occurrence* is unspecified, it defaults to 1. If *occurrence* is the null value, FINDSTR fails, the program terminates, and a run-time error message is produced.

Example

```
A="THIS":@FM:"IS":@FM:"A":@FM:"DYNAMIC":@FM:"ARRAY"
FINDSTR "IS" IN A SETTING FM,VM,SM ELSE ABORT
PRINT "FM=",FM
PRINT "VM=",VM
PRINT "SM=",SM
```

This is the program output:

```
FM=    1
VM=    1
SM=    1
```

Syntax

FIX (*number* [,*precision* [,*mode*]])

Description

Use the FIX function to convert a numeric value to a floating-point number with a specified precision. FIX lets you control the accuracy of computation by eliminating excess or unreliable data from numeric results. For example, a bank application that computes the interest accrual for customer accounts does not need to deal with credits expressed in fractions of cents. An engineering application needs to throw away digits that are beyond the accepted reliability of computations.

number is an expression that evaluates to the numeric value to be converted.

precision is an expression that evaluates to the number of digits of precision in the floating-point number. If you do not specify *precision*, the precision specified by the PRECISION statement is used. The default precision is 4.

mode is a flag that specifies how excess digits are handled. If *mode* is either 0 or not specified, excess digits are rounded off. If *mode* is anything other than 0, excess digits are truncated.

If *number* evaluates to the null value, null is returned.

Examples

The following example calculates a value to the default precision of 4:

```
REAL.VALUE = 37.73629273
PRINT FIX (REAL.VALUE)
```

This is the program output:

```
37.7363
```

The next example calculates the same value to two digits of precision. The first result is rounded off, the second is truncated:

```
PRINT FIX (REAL.VALUE, 2)
PRINT FIX (REAL.VALUE, 2, 1)
```

This is the program output:

```
37.74
37.73
```

FLUSH statement

Syntax

FLUSH *file.variable* {THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

The FLUSH statement causes all the buffers for a sequential I/O file to be written immediately. Normally, sequential I/O uses UNIX "stdio" buffering for input/output operations, and writes are not performed immediately.

file.variable specifies a file previously opened for sequential processing. If *file.variable* evaluates to the null value, the FLUSH statement fails, the program terminates, and a run-time error message is produced.

After the buffer is written to the file, the THEN statements are executed, and the ELSE statements are ignored. If THEN statements are not present, program execution continues with the next statement.

If the file cannot be written to or does not exist, the ELSE statements are executed; any THEN statements are ignored.

See the OPENSEQ, READSEQ, and WRITESEQ statements in Volume 2 for more information on sequential file processing.

Example

```
OPENSEQ 'FILE.E', 'RECORD1' TO FILE THEN
   PRINT "'FILE.E' OPENED FOR SEQUENTIAL PROCESSING"
END ELSE STOP
WEOFSEQ FILE
*
WRITESEQ 'NEW LINE' ON FILE THEN
   FLUSH FILE THEN
      PRINT "BUFFER FLUSHED"
   END ELSE PRINT "NOT FLUSHED"
ELSE ABORT
*
CLOSESEQ FILE
END
```

Syntax

FMT (*expression*, *format*)

expression format

Description

Use the FMT function or a format expression to format data for output. Any BASIC expression can be formatted for output by following it with a format expression.

expression evaluates to the numeric or string value to be formatted.

format is an expression that evaluates to a string of formatting codes. The syntax of the format expression is:

[*width*] [*fill*] *justification* [*edit*] [*mask*]

The format expression specifies the width of the output field, the placement of background or fill characters, line justification, editing specifications, and format masking.

If *expression* evaluates to the null value, null is returned. If *format* evaluates to null, the FMT function and the format operation fail.

width is an integer that specifies the size of the output field in which the value is to be justified. If you specify *mask*, you need not specify *width*. If you do not specify *mask*, *width* is required.

fill specifies the character to be used to pad entries when filling out the output field. *fill* is specified as a single character. The default fill character is a space. If you want to use a numeric character or the letter L, R, T, or Q as a fill character, you must enclose it in single quotation marks.

justification is required in one of the following forms.

Decimal notation:

L Left justification – Break on field length.

R Right justification – Break on field length.

T Text justification – Left justify and break on space.

U Left justification – Break on field length.

Exponential notation:

Q	Right justification – Break on field length.
QR	Right justification – Break on field length.
QL	Left justification

edit can be any of the following:

n[*m*]	Used with L, R, or T justification, *n* is the number of digits to display to the right of the decimal point, and *m* descales the value by *m* minus the current precision. Each can be a number from 0 through 9. You must specify *n* in order to specify *m*. If you do not specify *m*, *m* = 0 is assumed. If you do not specify *n*, *n* = *m* = 0 is assumed. Remember to account for the precision when you specify *m*. The default precision is 4. If you specify 0 for *n*, the value is rounded to the nearest integer. If the formatted value has fewer decimal places than *n*, output is padded with zeros to the *n*th decimal place. If the formatted value has more decimal places than *n*, the value is rounded to the *n*th decimal place. If you specify 0 for *m*, the value is descaled by the current precision (0 – current precision).
*n*E*m*	Used with Q, QR, or QL justification, *n* is the number of fractional digits, and *m* specifies the exponent. Each can be a number from 0 through 9.
n.m	Used with Q, QR, or QL justification, *n* is the number of digits preceding the decimal point, and *m* the number of fractional digits. Each can be a number from 0 through 9.
$	Prefixes a dollar sign to the value.
F	Prefixes a franc sign to the value.
,	Inserts commas after every thousand.
Z	Suppresses leading zeros. Returns an empty string if the value is 0. When used with the Q format, only the trailing fractional zeros are suppressed, and a 0 exponent is suppressed.
E	Surrounds negative numbers with angle brackets (< >).
C	Appends cr to negative numbers.
D	Appends db to positive numbers.

B	Appends db to negative numbers.
N	Suppresses a minus sign on negative numbers.
M	Appends a minus sign to negative numbers.
T	Truncates instead of rounding.

mask lets literals be intermixed with numerics in the formatted output field. The mask can include any combination of literals and the following three special format mask characters:

#*n*	Data is displayed in a field of *n* fill characters. A blank is the default fill character. It is used if the format string does not specify a fill character after the width parameter.
%*n*	Data is displayed in a field of *n* zeros.
**n*	Data is displayed in a field of *n* asterisks.

If you want to use numeric characters or any of the special characters as literals, you must escape the character with a backslash (\ ).

A #, %, or * character followed by digits causes the background fill character to be repeated *n* times. Other characters followed by digits cause those characters to appear in the output data *n* times.

mask can be enclosed in parentheses ( ) for clarity. If *mask* contains parentheses, you must include the whole mask in another set of parentheses. For example:

```
((###) ###-####)
```

You must specify either *width* or *mask* in the FMT function. You can specify both in the same function. When you specify *width*, the string is formatted according to the following rules:

If *string* is smaller than width *n*, it is padded with fill characters.

If *string* is larger than width *n*, a text mark (CHAR(251)) is inserted every *n*th character and each field is padded with the fill character to *width*.

The STATUS function reflects the result of *edit* as follows:

0	The edit code is successful.
1	The string expression is invalid.
2	The edit code is invalid.

See the STATUS function for more information.

REALITY Flavor

In REALITY flavor accounts, you can use conversion codes in format expressions.

Examples

Format Expression	Formatted Value
`Z=FMT("236986","R##-##-##")`	`Z=  23-69-86`
`X="555666898"` `X=FMT(X,"20*R2$,")`	`X=  *****$555,666,898.00`
`Y="DAVID"` `Y=FMT(Y,"10.L")`	`Y=  DAVID.....`
`V="24500"` `V=FMT(V,"10R2$Z")`	`V=  $24500.00`
`R=FMT(77777,"R#10")`	`R=  77777`
`B="0.12345678E1"` `B=FMT(B,"9*Q")`	`B=  *1.2346E0`
`PRINT 233779 "R"`	`233779`
`PRINT 233779 "R0"`	`233779`
`PRINT 233779 "R00"`	`2337790000`
`PRINT 233779 "R2"`	`233779.00`
`PRINT 233779 "R20"`	`2337790000.00`
`PRINT 233779 "R24"`	`233779.00`
`PRINT 233779 "R26"`	`2337.79`
`PRINT 2337.79 "R"`	`2337.79`
`PRINT 2337.79 "R0"`	`2338`
`PRINT 2337.79 "R00"`	`23377900`
`PRINT 2337.79 "R2"`	`2337.79`
`PRINT 2337.79 "R20"`	`23377900.00`

Format Expression	Formatted Value
`PRINT 2337.79 "R24"`	`2337.79`
`PRINT 2337.79 "R26"`	`23.38`

FMTS function

Syntax

FMTS (*dynamic.array, format*)

CALL –FMTS (*return.array, dynamic.array, format*)

CALL !FMTS (*return.array, dynamic.array, format*)

Description

Use the FMTS function to format elements of *dynamic.array* for output. Each element of the array is acted upon independently and is returned as an element in a new dynamic array.

format is an expression that evaluates to a string of formatting codes. The syntax of the format expression is:

[*width*] [*background*] *justification* [*edit*] [*mask*]

The format expression specifies the width of the output field, the placement of background or fill characters, line justification, editing specifications, and format masking. For complete syntax details, see the FMT function.

If *dynamic.array* evaluates to the null value, null is returned. If *format* evaluates to null, the FMTS function fails, the program terminates, and a run-time error message is produced.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

Syntax

FMUL (*number1*, *number2*)

CALL !FMUL (*return.array*, *number1*, *number2*)

Description

Use the FMUL function to perform floating-point multiplication on two numeric values. If either number evaluates to the null value, null is returned. *return.array* equates to *number1* multiplied by *number2*.

This function is provided for compatibility with existing software. You can also use the * operator to perform floating-point multiplication.

Example

```
PRINT FMUL(.234,.567)
```

This is the program output:

```
0.1327
```

FOLD function

Syntax

FOLD (*string*, *length*)

CALL !FOLD (*subdivided.string*, *length*, *string*)

Description

Use the FOLD function to divide a string into a number of substrings separated by field marks.

string is separated into substrings of length less than or equal to *length*. *string* is separated on blanks, if possible, otherwise it is separated into substrings of the specified length.

subdivided.string is the string you subdivide.

If *string* evaluates to the null value, null is returned. If *length* is less than 1, an empty string is returned. If *length* is the null value, the FOLD function fails, the program terminates, and a run-time error message is produced.

Examples

```
PRINT FOLD("THIS IS A FOLDED STRING.",5)
```

This is the program output:

```
THISFIS AFFOLDEFDFSTRINFG.
```

In the following example, the blanks are taken as substring delimiters, and as no substring exceeds the specified length of six characters, the output would be:

```
REDFMORANGEFMYELLOWFMGREENFMBLUEFMINDIGOFMVIOLET
```

Note that the field mark *replaces* the space in the string:

```
A="RED ORANGE YELLOW GREEN BLUE INDIGO VIOLET"
CALL !FOLD (RESULT,A,6)
PRINT RESULT
```

Syntax

FOOTING [ON *print.channel*] *footing*

Description

Use the FOOTING statement to specify the text and format of the footing to print at the bottom of each page of output.

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement). Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

footing is an expression that evaluates to the footing text and the control characters that specify the footing's format. You can use the following format control characters, enclosed in single quotation marks, in the footing expression:

C[*n*]	Prints footing line centered in a field of *n* blanks. If *n* is not specified, centers the line on the page.
D	Prints current date formatted as *dd mmm yyyy*.
G	Inserts gaps to format footings.
I	Resets page number, time, and date for PIOPEN flavor only.
Q	Allows the use of the] ^ and \ characters.
R[*n*]	Inserts the record ID left-justified in a field of *n* blanks.
S	Left-justified, inserted page number.
T	Prints current time and date formatted as *dd mmm yyyy hh:mm:ss*. Time is in 12-hour format with "am" or "pm" appended.
\	Prints current time and date formatted as *dd mmm yyyy hh:mm:ss*. Time is in 12-hour format with "am" or "pm" appended. Do not put the backslash inside single quotation marks.
L	Starts a new line.
]	Starts a new line. Do not put the right bracket inside single quotation marks.

P[*n*]	Prints current page number right-justified in a field of *n* blanks. The default value for *n* is 4.
^	Prints current page number right-justified in a field of *n* blanks. The default value for *n* is 4. Do not put the caret (^) inside single quotation marks.
N	Suppresses automatic paging.

Two single quotation marks ('') print one single quotation mark in footing text.

When the program is executed, the format control characters produce the specified results. You can specify multiple options in a single set of quotation marks.

If either *print.channel* or *footing* evaluates to the null value, the FOOTING statement fails, the program terminates, and a run-time error message is produced.

Pagination begins with page 1 and increments automatically on generation of each new page or upon encountering the PAGE statement.

Output to a terminal or printer is paged automatically. Use the N option in either a HEADING or a FOOTING statement to turn off automatic paging.

Using] ^ and \ in Footings

The characters] ^ and \ are control characters in headings and footings. To use these characters as normal characters, you must use the Q option and enclose the control character in double or single quotation marks. You only need to specify Q once in any heading or footing, but it must appear before any occurrence of the characters] ^ and \.

Formatting the Footing Text

The control character G (for gap) can be used to add blanks to text in footings to bring the width of a line up to device width. If G is specified once in a line, blanks are added to that part of the line to bring the line up to the device width. If G is specified at more than one point in a line, the blank characters are distributed as *evenly* as possible to those points.

See the following examples, in which the vertical bars represent the left and right margins:

Specification	Result
`"Hello there"`	`|Hello there  |`

Specification	Result
"'G'Hello there"	| Hello there|
"'G'Hello there'G'"	| Hello there |
"Hello'G'there"	|Hello there|
"'G'Hello'G'there'G'"	| Hello there |

The minimum gap size is 0 blanks. If a line is wider than the device width even when all the gaps are 0, the line wraps, and all gaps remain 0.

PIOPEN Flavor: Right-justified Overwriting Page Number

The control character P (for page) is right-justified at the point at which the P appears in the line. Only one character space is reserved for the number. If the number of digits exceeds 1, literal characters to the left of the initial position are overwritten. Normally you must enter a number of spaces to the left of the P to allow for the maximum page number to appear without overwriting other literal characters. For example, the statement:

```
FOOTING "This is page 'P' of 100000"
```

results in footings such as:

```
This is page 3 of 100000
This is pag333 of 100000
This is pa3333 of 100000
```

Left-justified Inserted Page Number

The control character S (for sequence number) is left-justified at the point where the S appears in the line. Only one character space is reserved for the number. If the number of digits exceeds 1, any text to the right is shifted right by the number of extra characters required. For example, the statement:

```
FOOTING "This is page 'S' of 100000"
```

results in footings such as:

```
This is page 3 of 100000
This is page 333 of 100000
This is page 3333 of 100000
```

PIOPEN Flavor: Resetting the Page Number and the Date

The control character I (for initialize) resets the page number to 1, and resets the date.

INFORMATION Flavor: Page Number Field

In an INFORMATION flavor account the default width of the page number field is the length of the page number. Use the *n* argument to P to set the field width of the page number. You can also include multiple P characters to specify the width of the page field, or you can include spaces in the text that immediately precedes a P option. For example, 'PPP' prints the page number right-justified in a field of three blanks.

Note: In all other flavors, 'PPP' prints three identical page numbers, each in the default field of four.

INFORMATION Flavor: Date Format

In an INFORMATION flavor account the default date format is *mm-dd-yy*, and the default time format is 24-hour style.

In PICK, IN2, REALITY, and IDEAL flavor accounts, use the HEADER.DATE option of the $OPTIONS statement to cause HEADER, FOOTER, and PAGE statements to behave as they do in INFORMATION flavor accounts.

Syntax

FOR *variable* = *start* TO *end* [STEP *increment*]
[*loop.statements*]
[CONTINUE]

[{WHILE | UNTIL} *expression*]
[*loop.statements*]
[CONTINUE]

NEXT [*variable*]

Description

Use the FOR statement to create a FOR...NEXT program loop. A program loop is a series of statements that execute repeatedly until the specified number of repetitions have been performed or until specified conditions are met.

variable is assigned the value of *start*, which is the initial value of the counter. *end* is the end value of the counter.

The *loop.statements* that follow the FOR clause execute until the NEXT statement is encountered. Then the counter is adjusted by the amount specified by the STEP clause.

At this point a check is performed on the value of the counter. If it is less than or equal to *end*, program execution branches back to the statement following the FOR clause and the process repeats. If it is greater than *end*, execution continues with the statement following the NEXT statement.

The WHILE condition specifies that as long as the WHILE expression evaluates to true, the loop continues to execute. When the WHILE expression evaluates to false, the loop ends, and program execution continues with the statement following the NEXT statement. If a WHILE or UNTIL expression evaluates to the null value, the condition is false.

The UNTIL condition specifies that the loop continues to execute only as long as the UNTIL expression evaluates to false. When the UNTIL expression evaluates to true, the loop ends and program execution continues with the statement following the NEXT statement.

expression can also contain a conditional statement. As *expression* you can use any statement that takes a THEN or an ELSE clause, but without the THEN or ELSE clause. When the conditional statement would execute the ELSE clause, *expression* evaluates to false; when the conditional statement would execute the THEN clause, *expression* evaluates to true. The LOCKED clause is not supported in this context.

You can use multiple WHILE and UNTIL clauses in a FOR...NEXT loop.

You can use the CONTINUE statement within FOR...NEXT to transfer control to the next iteration of the loop, from any point in the loop.

If STEP is not specified, *increment* is assumed to be 1. If *increment* is negative, the end value of the counter is less than the initial value. Each time the loop is processed, the counter is decreased by the amount specified in the STEP clause. Execution continues to loop until the counter is less than *end*.

The body of the loop is skipped if *start* is greater than *end*, and *increment* is not negative. If *start*, *end*, or *increment* evaluates to the null value, the FOR statement fails, the program terminates, and a run-time error message is produced.

Nested Loops

You can nest FOR...NEXT loops. That is, you can put a FOR...NEXT loop inside another FOR...NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before the NEXT statement for the outside loop.

If you omit the variables in the NEXT statement, the NEXT statement corresponds to the most recent FOR statement. If a NEXT statement is encountered without a previous FOR statement, an error occurs during compilation.

INFORMATION Flavor

In an INFORMATION flavor account the FOR variable is checked to see if it exceeds *end* before *increment* is added to it. That means that the value of the FOR variable does not exceed *end* at the termination of the loop. In IDEAL, PICK, IN2, and REALITY flavors the increment is made before the bound checking. In this case it is possible for *variable* to exceed *end*. Use the FOR.INCR.BEF option of the $OPTIONS statement to get IDEAL flavor behavior in an INFORMATION flavor account.

Examples

In the following example, the loop is executed 100 times or until control is transferred by one of the statements in the loop:

```
FOR VAR=1 TO 100
NEXT VAR
```

Here are more examples of FOR … NEXT loops:

Source Code	Program Output

`FOR X=1 TO 10`
`  PRINT "X= ",X`
`NEXT X`
	`X=  1`
	`X=  2`
	`X=  3`
	`X=  4`
	`X=  5`
	`X=  6`
	`X=  7`
	`X=  8`
	`X=  9`
	`X=  10`

`FOR TEST=1 TO 10 STEP 2`
`  PRINT "TEST= ":TEST`
`NEXT TEST`
	`TEST= 1`
	`TEST= 3`
	`TEST= 5`
	`TEST= 7`
	`TEST= 9`

`FOR SUB=50 TO 20 STEP -10`
`  PRINT 'VALUE IS ',SUB`
`NEXT`
	`VALUE IS  50`
	`VALUE IS  40`
	`VALUE IS  30`
	`VALUE IS  20`

`FOR A=1 TO 4`
`  FOR B=1 TO A`
`  PRINT "A:B= ",A:B`
`  NEXT B`
`NEXT A`
	`A:B=  11`
	`A:B=  21`
	`A:B=  22`
	`A:B=  31`
	`A:B=  32`
	`A:B=  33`
	`A:B=  41`
	`A:B=  42`
	`A:B=  43`
	`A:B=  44`

Source Code	Program Output

`PRINT 'LOOP 1 :'`
`SUM=0`
`FOR A=1 TO 10 UNTIL SUM>20`
`  SUM=SUM+A*A`
`  PRINT "SUM= ",SUM`
`NEXT`
	`LOOP 1 :`
	`SUM=  1`
	`SUM=  5`
	`SUM=  14`
	`SUM=  30`

`PRINT 'LOOP 2 :'`
`*`
`Y=15`
`Z=0`
`FOR X=1 TO 20 WHILE Z<Y`
`  Z=Z+X`
`  PRINT "Z= ",Z`
`NEXT X`
	`LOOP 2 :`
	`Z=  1`
	`Z=  3`
	`Z=  6`
	`Z=  10`
	`Z=  15`

FORMLIST statement

Syntax

FORMLIST [*variable*] [TO *list.number*] [ON ERROR *statements*]

Description

The FORMLIST statement is the same as the SELECT statement.

FSUB function

Syntax

FSUB (*number1*, *number2*)

CALL !FSUB (*result*, *number1*, *number2*)

Description

Use the FSUB function to perform floating-point subtraction on two numeric values. *number2* is subtracted from *number1*. If either number evaluates to the null value, null is returned. *result* equates to *number1* minus *number2*.

This function is provided for compatibility with existing software. You can also use the – operator to perform floating-point subtraction.

Example

```
PRINT FSUB(.234,.567)
```

This is the program output:

```
-0.333
```

Syntax

FUNCTION [*name*] [([MAT] *variable* [, [MAT] *variable* ...])]

Description

Use the FUNCTION statement to identify a user-written function and to specify the number and names of the arguments to be passed to it. The FUNCTION statement must be the first noncomment line in the user-written function. A user-written function can contain only one FUNCTION statement.

name is specified for documentation purposes; it need not be the same as the function name or the name used to reference the function in the calling program. *name* can be any valid variable name.

variable is an expression that passes values between the calling programs and the function. *variables* are the formal parameters of the user-written function. When actual parameters are specified as arguments to a user-written function, the actual parameters are referenced by the formal parameters so that calculations performed in the user-written function use the actual parameters.

Separate *variables* by commas. Up to 254 variables can be passed to a user-written function. To pass an array, you must precede the array name with the keyword MAT. When a user-written function is called, the calling function must specify the same number of variables as are specified in the FUNCTION statement.

An extra variable is hidden so that the user-written function can use it to return a value. An extra variable is retained by the user-written function so that a value is returned by the RETURN (*value*) statement. This extra variable is reported by the MAP and MAKE.MAP.FILE commands. If you use the RETURN statement in a user-written function and you do not specify a value to return, an empty string is returned by default.

The program that calls a user-written function must contain a DEFFUN statement that defines the user-written function before it uses it. The user-written function must be cataloged in either a local catalog or the system catalog, or it must be a record in the same object file as the calling program.

If the user-defined function recursively calls itself within the function, a DEFFUN statement must precede it in the user-written function.

Examples

The following user-defined function SHORT compares the length of two arguments and returns the shorter:

```
FUNCTION SHORT(A,B)
AL = LEN(A)
BL = LEN(B)
IF AL < BL THEN RESULT = A ELSE RESULT = B
RETURN(RESULT)
```

The following example defines a function called MYFUNC with the arguments or formal parameters A, B, and C. It is followed by an example of the DEFFUN statement declaring and using the MYFUNC function. The actual parameters held in X, Y, and Z are referenced by the formal parameters A, B, and C so that the value assigned to T can be calculated.

```
FUNCTION MYFUNC(A, B, C)
Z = ...
RETURN (Z)
    .
    .
    .
END

DEFFUN MYFUNC(X, Y, Z)
T = MYFUNC(X, Y, Z)
END
```

Syntax

GES (*array1*, *array2*)

CALL –GES (*return.array*, *array1*, *array2*)

CALL !GES (*return.array*, *array1*, *array2*)

Description

Use the GES function to test if elements of one dynamic array are greater than or equal to corresponding elements of another dynamic array.

Each element of *array1* is compared with the corresponding element of *array2*. If the element from *array1* is greater than or equal to the element from *array2*, a 1 is returned in the corresponding element of a new dynamic array. If the element from *array1* is less than the element from *array2*, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, the undefined element is evaluated as empty, and the comparison continues.

If either element of a corresponding pair is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

GET statements

Syntax

UNIX

GET[X] *read.var*[,*length*] [SETTING *read.count*] FROM *device*
[UNTIL *eop.char.list*] [RETURNING *last.char.read*]
[WAITING *seconds*] [THEN *statements*] [ELSE *statements*]

Windows NT

GET[X] *read.var*[,*length*] [SETTING *read.count*] FROM *device*
[UNTIL *eop.char.list*] [RETURNING *last.char.read*]
[THEN *statements*] [ELSE *statements*]

Description

Use GET statements to read a block of data from an input stream associated with a device, such as a serial line or terminal. The device must be opened with the OPENDEV or OPENSEQ statement. Once the device has been opened, the GET statements read data from the device. The GET statements do not perform any pre- or postprocessing of the data stream, nor do they control local echo characteristics. These aspects of terminal control are handled either by the application or by the device driver. The behavior of certain devices can be managed through the TTYSET/TTYGET interface.

The WAITING clause is not supported on Windows NT.

Use the GETX statement to return the characters in ASCII hexadecimal format. For example, the sequence of 8-bit character “abcde” is returned as the character string “6162636465”. However, the value returned in the *last.char.read* variable is in standard ASCII character form.

read.var is the variable into which the characters read from *device* are stored. If no data is read, *read.var* is set to the empty string.

length is the expression evaluating to the number of characters read from the data stream; if *length* and *timeout* are not specified, the default length is 1. If *length* is not specified, but an *eop.char.list* value is included, no length limit is imposed on the input.

read.count is the variable that records the actual count of characters read and stored in *read.var*. This may differ from *length* when a timeout condition occurs or when a recognized end-of-packet character is detected.

device is a valid file variable resulting from a successful OPENDEV or OPENSEQ statement. This is the handle to the I/O device that supplies the data stream for the operation of the GET statements.

eop.char.list is an expression that evaluates to a recognized end-of-packet delimiters. The GET operation terminates if a valid end-of-packet character is encountered in the data stream before the requested number of characters is read.

last.char.read is a variable that stores the last character read by the GET operation. If no data is read, *read.var* is set to the empty string. If the input terminated due to the maximum number of characters being read or because of a timeout condition, an empty string is returned.

seconds specifies the number of seconds the program should wait before the GET operation times out.

Terminating Conditions

GET statements read data from the device's input stream until the first terminating condition is encountered. The following table lists the possible terminating conditions:

GET Statements Terminating Conditions

Condition	Description
Requested read length has been satisfied	The read is fully satisfied. *read.var* contains the characters read, and *last.char.read* contains an empty string. Program control passes to the THEN clause if present. Note that the default requested read length is one character unless an end-of-packet value has been selected (in which case, no length limit is used).

GET Statements Terminating Conditions (Continued)

Condition	Description
Recognized end-of-packet character has been processed	The read is terminated by a special application-defined character in the data stream. The data read to this point, excluding the end-of-packet character, is stored in *read.var*. The end-of-packet character is stored in *last.char.read*. Program control passes to the THEN clause if present. This terminating condition is only possible if the UNTIL clause has been specified. If there is no UNTIL clause, no end-of-packet characters are recognized.
Timeout limit has expired	The read could not be satisfied within the specified time limitation. If no characters have been read, *read.var* and *last.char.read* are set to the empty string, and *read.count* is set to 0. The system status code is set to 0 and may be checked with the STATUS function. Control passes to the ELSE clause if present. This condition is only possible if the WAITING clause is specified. In the absence of a WAITING clause, the application waits until one of the other terminating conditions is met.
Device input error	An unrecoverable error occurred on the device. Unrecoverable errors can include EOF conditions and operating system reported I/O errors. In this case, the data read to this point is stored in *read.var*, and the empty string is stored in *last.char.read*. If no characters have been read, *read.var* and *last.char.read* are set to the empty string, and *read.count* is set to 0. The system status code is set to a nonzero value and may checked with the STATUS function. Control passes to the ELSE clause if present.

Note: Under all termination conditions, *read.count* is set to the number of characters read from the input data stream.

THEN and ELSE Clauses

For GET statements, the THEN and ELSE clauses are optional. They have different meanings and produce different results, depending on the conditions specified for terminal input.

The following rules apply only if the THEN or ELSE clauses are specified:

- If the UNTIL clause is used without a WAITING clause or an expected length, the GET statement behaves normally. The program waits indefinitely until a termination character is read, then executes the THEN clause. The ELSE clause is never executed.
- If the WAITING clause is used, the GET statement behaves normally, and the ELSE clause is executed only if the number of seconds for timeout has elapsed. If the input terminates for any other reason, it executes the THEN clause.
- If the WAITING clause is not used and there is a finite number of characters to expect from the input, then only the type-ahead buffer is examined for input. If the type-ahead buffer contains the expected number of characters, it executes the THEN clause; otherwise it executes the ELSE clause. Note that if the type-ahead feature is turned off, the ELSE clause is always executed.
- In a special case, the ELSE clause is executed if the line has not been attached before executing the GET statement.

In summary, unless the WAITING clause is used, specifying the THEN and ELSE clauses causes the GET statement to behave like an INPUTIF…FROM statement. The exception to this is the UNTIL clause without a maximum length specified, in which case the GET statement behaves normally and the ELSE clause is never used.

Example

The following code fragment shows how the GET statement reads a number of data buffers representing a transaction message from a device:

```
DIM SAVEBUFFER(10)
SAVELIMIT = 10
OPENDEV "TTY10" TO TTYLINE ELSE STOP "CANNOT OPEN TTY10"
I = 1
LOOP
```

```
       GET BUFFER,128 FROM TTYLINE UNTIL CHAR(10) WAITING 10
       ELSE
          IF STATUS()
          THEN PRINT "UNRECOVERABLE ERROR DETECTED ON DEVICE,
"IM SAVEBUFFER(10)
SAVELIMIT = 10
OPENDEV "TTY10" TO TTYLINE ELSE STOP "CANNOT OPEN TTY10"
I = 1
LOOP
       GET BUFFER,128 FROM TTYLINE UNTIL CHAR(10)
WAITING 10
       ELSE
          IF STATUS()
          THEN PRINT "UNRECOVERABLE ERROR DETECTED ON DEVICE,":
          ELSE PRINT "DEVICE TIMEOUT HAS OCCURRED, ":
          PRINT "TRANSACTION CANNOT BE COMPLETED."
          STOP
       END
   WHILE BUFFER # "QUIT" DO
       IF I > SAVELIMIT
       THEN
          SAVELIMIT += 10
          DIM SAVEBUFFER(SAVELIMIT)
       END
       SAVEBUFFER(I) = BUFFER
       I += 1
REPEAT
":
          ELSE PRINT "DEVICE TIMEOUT HAS OCCURRED, ":
          PRINT "TRANSACTION CANNOT BE COMPLETED."
          STOP
       END
   WHILE BUFFER # "QUIT" DO
       IF I > SAVELIMIT
       THEN
          SAVELIMIT += 10
          DIM SAVEBUFFER(SAVELIMIT)
       END
       SAVEBUFFER(I) = BUFFER
       I += 1
REPEAT
```

GETX statement

Use the GETX statement to read a block of data from an input stream and return the characters in ASCII hexadecimal format. For details, see the GET statements.

GET(ARG.) statement

Syntax

GET(ARG. [,*arg#*]) *variable* [THEN *statements*] [ELSE *statements*]

Description

Use the GET(ARG.) statement to retrieve the next command line argument. The command line is delimited by blanks, and the first argument is assumed to be the first word after the program name. When a cataloged program is invoked, the argument list starts with the second word in the command line.

Blanks in quoted strings are not treated as delimiters and the string is treated as a single argument. For example, "54 76" returns 54 76.

arg# specifies the command line argument to retrieve. It must evaluate to a number. If *arg#* is not specified, the next command line argument is retrieved. The retrieved argument is assigned to *variable*.

THEN and ELSE statements are both optional. The THEN clause is executed if the argument is found. The ELSE clause is executed if the argument is not found. If the argument is not found and no ELSE clause is present, *variable* is set to an empty string.

If no *arg#* is specified or if *arg#* evaluates to 0, the argument to the right of the last argument retrieved is assigned to *variable*. The GET statement fails if *arg#* evaluates to a number greater than the number of command line arguments or if the last argument has been assigned and a GET with no *arg#* is used. To move to the beginning of the argument list, set *arg#* to 1.

If *arg#* evaluates to the null value, the GET statement fails, the program terminates, and a run-time error message is produced.

Example

In the following example, the command is:

```
RUN BP PROG ARG1 ARG2 ARG3
```

and the program is:

```
A=5;B=2
GET(ARG.)FIRST
GET(ARG.,B)SECOND
GET(ARG.)THIRD
```

```
GET(ARG.,1)FOURTH
GET(ARG.,A-B)FIFTH
PRINT FIRST

PRINT SECOND
PRINT THIRD
PRINT FOURTH
PRINT FIFTH
```

This is the program output:

```
ARG1
ARG2
ARG3
ARG1
ARG3
```

If the command line is changed to RUN PROG, the system looks in the file PROG for the program with the name of the first argument. If PROG is a cataloged program, the command line would have to be changed to PROG ARG1 ARG2 ARG3 to get the same results.

GETLIST statement

Syntax

GETLIST *listname* [TO *list.number*] [SETTING *variable*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the GETLIST statement to activate a saved select list so that a READNEXT statement can use it.

listname is an expression that evaluates to the form:

record.ID

or:

record.ID account.name

record.ID is the record ID of a select list in the &SAVEDLISTS& file. If *account.name* is specified, the &SAVEDLISTS& file of that account is used instead of the one in the local account.

If *listname* evaluates to the null value, the GETLIST statement fails, the program terminates, and a run-time error message is produced.

The TO clause puts the list in a select list numbered 0 through 10. If *list.number* is not specified, the list is saved as select list 0.

The SETTING clause assigns the count of the elements in the list to *variable*. The system variable @SELECTED is also assigned this count whether or not the SETTING clause is used. If the list is retrieved successfully, even if the list is empty, the THEN statements execute; if not, the ELSE statements execute.

PICK, REALITY, and IN2 Flavors

PICK, REALITY, and IN2 flavor accounts store select lists in list variables instead of numbered select lists. In those accounts, and in programs that use the VAR.SELECT option of the $OPTIONS statement, the syntax of the GETLIST statement is:

GETLIST *listname* [TO *list.variable*] [SETTING *variable*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Syntax

GETREM (*dynamic.array*)

Description

Use the GETREM function after the execution of a REMOVE statement, a REMOVE function, or a REVREMOVE statement, to return the numeric value for the character position of the pointer associated with *dynamic.array*.

dynamic.array evaluates to the name of a variable containing a dynamic array.

The returned value is an integer. The integer returned is one-based, not zero-based. If no REMOVE statements have been executed on *dynamic.array*, 1 is returned. At the end of *dynamic.array*, GETREM returns the length of dynamic array plus 1. The offset returned by GETREM indicates the first character of the next dynamic array element to be removed.

Example

```
DYN = "THIS":@FM:"HERE":@FM:"STRING"
REMOVE VAR FROM DYN SETTING X
PRINT GETREM(DYN)
```

This is the program output:

```
5
```

GOSUB statement

Syntax

GOSUB *statement.label* [:]

GO SUB *statement.label* [:]

Description

Use the GOSUB statement to transfer program control to an internal subroutine referenced by *statement.label*. A colon (:) is optional in GOSUB statements, even though it is required after nonnumeric statement labels at the beginning of program lines.

Use the RETURN statement at the end of the internal subroutine referenced by the GOSUB statement, to transfer program control to the statement following the GOSUB statement.

Use the RETURN TO statement at the end of an internal subroutine to transfer control to a location in the program other than the line following the GOSUB statement. In this case, use *statement.label* to refer to the target location.

Be careful with the RETURN TO statement, because all other GOSUBs or CALLs active when the GOSUB is executed remain active, and errors can result.

A program can call a subroutine any number of times. A subroutine can also be called from within another subroutine; this process is called nesting subroutines. You can nest up to 256 GOSUB calls.

Subroutines can appear anywhere in the program but should be readily distinguishable from the main program. To prevent inadvertent entry into the subroutine, precede it with a STOP, END, or GOTO statement that directs program control around the subroutine.

Example

```
VAR='ABKL1234'
FOR X=1 TO LEN(VAR)
   Y=VAR[X,1]
   GOSUB 100
NEXT X
STOP
100*
IF Y MATCHES '1N' THEN RETURN TO 200
```

```
PRINT 'ALPHA CHARACTER IN POSITION ',X
RETURN
200*

PRINT 'NUMERIC CHARACTER IN POSITION ',X
STOP
```

This is the program output:

```
ALPHA CHARACTER IN POSITION  1
ALPHA CHARACTER IN POSITION  2
ALPHA CHARACTER IN POSITION  3
ALPHA CHARACTER IN POSITION  4
NUMERIC CHARACTER IN POSITION          5
```

GOTO statement

Syntax

GO[TO] *statement.label* [:]

GO TO *statement.label* [:]

Description

Use the GOTO statement to transfer program control to the statement specified by *statement.label*. A colon (:) is optional in GOTO statements.

If the statement referenced is an executable statement, that statement and those that follow are executed. If it is a nonexecutable statement, execution proceeds at the first executable statement encountered after the referenced statement.

Example

```
X=80
GOTO 10
STOP
*
10*
IF X>20 THEN GO 20 ELSE STOP
*
20*
PRINT 'AT LABEL 20'
GO TO CALCULATE:
STOP
*
CALCULATE:
PRINT 'AT LABEL CALCULATE'
```

This is the program output:

```
AT LABEL 20
AT LABEL CALCULATE
```

Syntax

GROUP (*string*, *delimiter*, *occurrence* [,*num.substr*])

Description

Use the GROUP function to return one or more substrings located between specified delimiters in *string*.

delimiter evaluates to any character, including field mark, value mark, and subvalue marks. It delimits the start and end of the substring. If *delimiter* evaluates to more than one character, only the first character is used. Delimiters are not returned with the substring.

occurrence specifies which occurrence of the delimiter is to be used as a terminator. If *occurrence* is less than 1, 1 is assumed.

num.substr specifies the number of delimited substrings to return. If the value of *num.substr* is an empty string or less than 1, 1 is assumed. When more than one substring is returned, delimiters are returned along with the successive substrings.

If either *delimiter* or *occurrence* is not in the string, an empty string is returned, unless *occurrence* specifies 1. If *occurrence* is 1 and *delimiter* is not found, the entire string is returned. If *delimiter* is an empty string, the entire string is returned.

If *string* evaluates to the null value, null is returned. If *string* contains CHAR(128) (that is, @NULL.STR), it is treated like any other character in a string. If *delimiter*, *occurrence*, or *num.substr* evaluates to the null value, the GROUP function fails, the program terminates, and a run-time error message is produced.

The GROUP function works identically to the FIELD function.

Examples

```
D=GROUP("###DHHH#KK","#",4)
PRINT "D= ",D
```

The variable D is set to DHHH because the data between the third and fourth occurrence of the delimiter # is DHHH.

```
REC="ACADABA"
E=GROUP(REC,"A",2)
PRINT "E= ",E
```

The variable E is set to "C".

```
VAR="?"
Z=GROUP("A.1234$$$$&&",VAR,3)
PRINT "Z= ",Z
```

Z is set to an empty string since "?" does not appear in the string.

```
Q=GROUP("+1+2+3ABAC","+",2,2)
PRINT "Q= ",Q
```

Q is set to "1+2" since two successive fields were specified to be returned after the second occurrence of "+".

This is the program output:

```
D=        DHHH
E=        C
Z=
Q=        1+2
```

Syntax

GROUPSTORE *new.string* IN *string* USING *start*, *n* [,*delimiter*]

Description

Use the GROUPSTORE statement to modify character strings by inserting, deleting, or replacing fields separated by specified delimiters.

new.string is an expression that evaluates to the character string to be inserted in *string*.

string is an expression that evaluates to the character string to be modified.

delimiter evaluates to any single ASCII character, including field, value, and subvalue marks. If you do not specify *delimiter*, the field mark is used.

start evaluates to a number specifying the starting field position. Modification begins at the field specified by *start*. If *start* is greater than the number of fields in *string*, the required number of empty fields is generated before the GROUPSTORE statement is executed.

n specifies the number of fields of *new.string* to insert in *string*. *n* determines how the GROUPSTORE operation is executed. If *n* is positive, *n* fields in *string* are replaced with the first *n* fields of *new.string*. If *n* is negative, *n* fields in *string* are replaced with all the fields in *new.string*. If *n* is 0, all the fields in *new.string* are inserted in *string* before the field specified by *start*.

If *string* evaluates to the null value, null is returned. If *new.string*, *start*, *n*, or *delimiter* is null, the GROUPSTORE statement fails, the program terminates, and a run-time error message is produced.

Example

```
Q='1#2#3#4#5'
GROUPSTORE "A#B" IN Q USING 2,2,"#"
PRINT "TEST1= ",Q
*
Q='1#2#3#4#5'
GROUPSTORE "A#B" IN Q USING 2,-2,"#"
PRINT "TEST2= ",Q
*
Q='1#2#3#4#5'
GROUPSTORE "A#B" IN Q USING 2,0,"#"
```

```
PRINT "TEST3= ",Q
*

Q='1#2#3#4#5'
GROUPSTORE "A#B#C#D" IN Q USING 1,4,"#"
PRINT "TEST4= ",Q
*
Q='1#2#3#4#5'
GROUPSTORE "A#B#C#D" IN Q USING 7,3,"#"
PRINT "TEST5= ",Q
```

This is the program output:

```
TEST1=   1#A#B#4#5
TEST2=   1#A#B#4#5
TEST3=   1#A#B#2#3#4#5
TEST4=   A#B#C#D#5
TEST5=   1#2#3#4#5##A#B#C
```

Syntax

GTS (*array1*, *array2*)

CALL –GTS (*return.array*, *array1*, *array2*)

CALL !GTS (*return.array*, *array1*, *array2*)

Description

Use the GTS function to test if elements of one dynamic array are greater than elements of another dynamic array.

Each element of *array1* is compared with the corresponding element of *array2*. If the element from *array1* is greater than the element from *array2*, a 1 is returned in the corresponding element of a new dynamic array. If the element from *array1* is less than or equal to the element from *array2*, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, the undefined element is evaluated as an empty string, and the comparison continues.

If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

HEADING statement

Syntax

HEADING [ON *print.channel*] *heading*

HEADINGE [ON *print.channel*] *heading*

HEADINGN [ON *print.channel*] *heading*

Description

Use the HEADING statement to specify the text and format of the heading to print at the top of each page of output.

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement). Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

heading is an expression that evaluates to the heading text and the control characters that specify the heading's format. You can use the following format control characters, enclosed in single quotation marks, in the heading expression:

C[*n*]	Prints heading line centered in a field of *n* blanks. If *n* is not specified, centers the line on the page.
D	Prints current date formatted as *dd mmm yyyy*.
T	Prints current time and date formatted as *dd mmm yyyy hh:mm:ss*. Time is in 12-hour format with "am" or "pm" appended.
\	Prints current time and date formatted as *dd mmm yyyy hh:mm:ss*. Time is in 12-hour format with "am" or "pm" appended. Do not put the backslash inside single quotation marks.
G	Inserts gaps to format headings.
I	Resets page number, time, and date for PIOPEN flavor only.
Q	Allows the use of the] ^ and \ characters.
R[*n*]	Inserts the record ID left-justified in a field of *n* blanks.
L	Starts a new line.

]	Starts a new line. Do not put the right bracket inside single quotation marks.
N	Suppresses automatic paging.
P[*n*]	Prints current page number right-justified in a field of *n* blanks. The default value for *n* is 4.
S	Left-justified, inserted page number.
^	Prints current page number right-justified in a field of *n* blanks. The default value for *n* is 4. Do not put the caret inside single quotation marks.

Two single quotation marks (' ') print one single quotation mark in heading text.

When the program is executed, the format control characters produce the specified results. You can specify multiple options in a single set of quotation marks.

If either *print.channel* or *heading* evaluates to the null value, the HEADING statement fails, the program terminates, and a run-time error message is produced.

Pagination begins with page 1 and increments automatically on generation of each new page or upon encountering the PAGE statement.

Output to a terminal or printer is paged automatically. Use the N option in either a HEADING or a FOOTING statement to turn off automatic paging.

HEADINGE and HEADINGN Statements

The HEADINGE statement is the same as the HEADING statement with the $OPTIONS HEADER.EJECT selected. HEADINGE causes a page eject with the HEADING statement. Page eject is the default for INFORMATION flavor accounts.

The HEADINGN statement is the same as the HEADING statement with the $OPTIONS –HEADER.EJECT selected. HEADINGN suppresses a page eject with the HEADING statement. The page eject is suppressed in IDEAL, PICK, REALITY, and IN2 flavor accounts.

Using] ^ and \ in Headings

The characters] ^ and \ are control characters in headings and footings. To use these characters as normal characters, you must use the Q option and enclose the control character in double or single quotation marks. You only need to specify Q

once in any heading or footing, but it must appear before any occurrence of the characters] ^ and \.

Formatting the Heading Text

The control character G (for gap) can be used to add blanks to text in headings to bring the width of a line up to device width. If G is specified once in a line, blanks are added to that part of the line to bring the line up to the device width. If G is specified at more than one point in a line, the space characters are distributed as *evenly* as possible to those points. See the following examples, in which the vertical bars represent the left and right margins:

Specification	Result
`"Hello there"`	`|Hello there|`
`"'G'Hello there"`	`|  Hello there|`
`"'G'Hello there'G'"`	`|  Hello there  |`
`"Hello'G'there"`	`|Hello  there|`
`"'G'Hello'G'there'G'"`	`|  Hello  there  |`

The minimum gap size is 0 blanks. If a line is wider than the device width even when all the gaps are 0, the line wraps, and all gaps remain 0.

Left-justified Inserted Page Number

The control character S (for sequence number) is left-justified at the point where the S appears in the line. Only one character space is reserved for the number. If the number of digits exceeds 1, any text to the right is shifted right by the number of extra characters required. For example, the statement:

```
HEADING "This is page 'S' of 100000"
```

results in headings such as:

```
This is page 3 of 100000
This is page 333 of 100000
This is page 3333 of 100000
```

INFORMATION Flavor

Page Number Field. In an INFORMATION flavor account the default width of the page number field is the length of the page number. Use the *n* argument to P to

set the field width of the page number. You can also include multiple P characters to specify the width of the page field, or you can include blanks in the text that immediately precedes a P option. For example, 'PPP' prints the page number right-justified in a field of three blanks.

Note: In all other flavors, 'PPP' prints three identical page numbers, each in the default field of four.

Date Format. In an INFORMATION flavor account the default date format is *mm-dd-yy*, and the default time format is 24-hour style.

In PICK, IN2, REALITY, and IDEAL flavor accounts, use the HEADER.DATE option of the $OPTIONS statement to cause HEADER, FOOTER, and PAGE statements to behave as they do in INFORMATION flavor accounts.

PIOPEN Flavor

Right-justified Overwriting Page Number. The control character P (for page) is right-justified at the point at which the P appears in the line. Only one character space is reserved for the number. If the number of digits exceeds 1, literal characters to the left of the initial position are overwritten. Normally you must enter a number of blanks to the left of the P to allow for the maximum page number to appear without overwriting other literal characters. For example, the statement:

```
HEADING "This is page 'P' of 100000"
```

results in headings such as:

```
This is page 3 of 100000
This is pag333 of 100000
This is pa3333 of 100000
```

Resetting the Page Number and the Date. The control character I (for initialize) resets the page number to 1, and resets the date.

Example

```
HEADING "'C' LIST PRINTED:  'D'"
FOR N=1 TO 10
   PRINT "THIS IS ANOTHER LINE"
NEXT
```

This is the program output:

```
                 LIST PRINTED:  04 Jun 1994
   THIS IS ANOTHER LINE
   THIS IS ANOTHER LINE
   THIS IS ANOTHER LINE
   THIS IS ANOTHER LINE
   THIS IS ANOTHER LINE
   THIS IS ANOTHER LINE
   THIS IS ANOTHER LINE
   THIS IS ANOTHER LINE
   THIS IS ANOTHER LINE
   THIS IS ANOTHER LINE
```

Syntax

HUSH { ON | OFF | *expression* } [SETTING *status*]

Description

Use the HUSH statement to suppress the display of all output normally sent to a terminal during processing. HUSH also suppresses output to a COMO file or TANDEM display.

SETTING *status* sets the value of a variable to the value of the HUSH state before the HUSH statement was executed. It can be used instead of the STATUS function to save the state so that it can be restored later. STATUS has a value of 1 if the previous state was HUSH ON or a value of 0 if the previous state was HUSH OFF.

You might use this statement when you are transmitting information over phone lines or when you are sending data to a hard-copy terminal. Both these situations result in slower transmission speeds. The unnecessary data display makes the task even slower.

HUSH acts as a toggle. If it is used without a qualifier, it changes the current state.

Do not use this statement to shut off output display unless you are sure the display is unnecessary. When you use HUSH ON, all output is suppressed including error messages and requests for information.

Values Returned by the STATUS Function

The previous state is returned by the STATUS function. If terminal output was suppressed prior to execution of the HUSH statement, the STATUS function returns a 1. If terminal output was enabled before execution of the HUSH statement, the STATUS function returns a 0.

Example

In the following example, terminal output is disabled with the HUSH statement and the previous state was saved in the variable USER.HUSH.STATE.

After executing some other statements, the program returns the user's process to the same HUSH state as it was in previous to the execution of the first HUSH statement:

```
HUSH ON
USER.HUSH.STATE = STATUS()
...
HUSH USER.HUSH.STATE
```

The example could have been written as follows:

```
HUSH ON SETTING USER.HUSH.STATE
    .
    .
    .
HUSH USER.HUSH.STATE
```

Syntax

ICHECK (*dynamic.array* [,*file.variable*] ,*key* [,*column#*])

Description

Use the ICHECK function to check if data you intend to write to an SQL table violates any SQL integrity constraints. ICHECK verifies that specified data and primary keys satisfy the defined SQL integrity constraints for an SQL table.

dynamic.array is an expression that evaluates to the data you want to check against any integrity constraints.

file.variable specifies an open file. If *file.variable* is not specified, the default file variable is assumed (for more information on default files, see the OPEN statement in Volume 2).

key is an expression that evaluates to the primary key you want to check against any integrity constraints.

column# is an expression that evaluates to the number of the column in the table whose data is to be checked. If you do not specify *column#*, all columns in the file are checked. Column 0 specifies the primary key (record ID).

If *dynamic.array*, *file.variable*, *key*, or *column#* evaluates to the null value, the ICHECK function fails, the program terminates, and a run-time error message is produced.

You might use the ICHECK function to limit the amount of integrity checking that is done and thus improve performance. If you do this, however, you are assuming responsibility for data integrity. For example, you might want to use ICHECK with a program that changes only a few columns in a file. To do this, turn off the OPENCHK configurable parameter, open the file with the OPEN statement rather than the OPENCHECK statement, and use the ICHECK function before you write the updated record to verify, for each column you are updating, that you are not violating the table's integrity checks.

If the ON UPDATE clause of a referential constraint specifies an action, ICHECK always validates data being written to the referenced table; it does not check the referencing table. Therefore, ICHECK can succeed, but when the actual write is done, it can have a constraint failure while attempting to update the referencing table. If the referential constraint does not have an ON UPDATE clause, or if these clauses specify NO ACTION, the referencing table is checked to ensure that no row in it contains the old value of the referenced column.

ICHECK function

ICHECK returns a dynamic array of three elements separated by field marks:

*error.code*F*column#*F*constraint*

error.code A code that indicates the type of failure. Error codes can be any of the following:

0 No failure

1 SINGLEVALUED failure

2 NOT NULL failure

3 NOT EMPTY failure

4 ROWUNIQUE failure (including single-column association KEY)

5 UNIQUE (column constraint) failure

6 UNIQUE (table constraint) failure

7 Association KEY ROWUNIQUE failure when association has multiple KEY fields (not implemented in Release 8)

8 CHECK constraint failure

9 Primary key has too many parts

10 Referential constraint failure

column# The number of the column where the failure occurred. If any part of a primary key fails, 0 is returned.

constraint This element is returned only for CHECK constraints. If the constraint has a name, the name is returned. If it does not have a name, the text of the CHECK constraint is returned.

If the record violates more than one integrity constraint, ICHECK returns a dynamic array only for the first constraint that causes a failure.

The ICHECK function is advisory only. That is, if two programs try to write the same data to the same column defined as UNIQUE (see error 5), an ICHECK in the first program may pass. If the second program writes data to the file before the first program writes its ICHECKed data, the first program's write fails even though the ICHECK did not fail.

Syntax

ICONV (*string*, *conversion*)

Description

Use the ICONV function to convert *string* to a specified internal storage format. *string* is an expression that evaluates to the string to be converted. *conversion* is an expression that evaluates to one or more valid conversion codes, separated by value marks (ASCII 253).

string is converted to the internal format specified by conversion. If multiple codes are used, they are applied from left to right. The first conversion code converts the value of *string*. The second conversion code converts the output of the first conversion, and so on.

If *string* evaluates to the null value, null is returned. If *conversion* evaluates to the null value, the ICONV function fails, the program terminates, and a run-time error message is produced.

The STATUS function reflects the result of the conversion:

0 The conversion is successful.

1 *string* is invalid. An empty string is returned, unless *string* is the null value, in which case null is returned.

2 *conversion* is invalid.

3 Successful conversion of possibly invalid data.

See the STATUS function in Volume 2 for more information.

For information about converting strings to an external format, see the OCONV function.

Examples

The following are examples of date conversions:

Source Line	Converted Value
`DATE=ICONV("02-23-85","D")`	`6264`
`DATE=ICONV("30/9/67","DE")`	`-92`

Source Line	Converted Value
`DATE=ICONV("6-10-85","D")`	`6371`
`DATE=ICONV("19850625","D")`	`6386`
`DATE=ICONV("85161","D")`	`6371`

The following is an example of a time conversion:

Source Line	Converted Value
`TIME=ICONV("9AM","MT")`	`32400`

The following are examples of hex, octal, and binary conversions:

Source Line	Converted Value
`HEX=ICONV("566D61726B","MX0C")`	`Vmark`
`OCT=ICONV("3001","MO")`	`1537`
`BIN=ICONV(1111,"MB")`	`15`

The following are examples of masked decimal conversions:

Source Lines	Converted Value
`X=4956.00` `X=ICONV(X,"MD2")`	`495600`
`X=563.888` `X=ICONV(X,"MD0")`	`-564`
`X=ICONV(1988.28,"MD24")`	`19882800`

Syntax

ICONVS (*dynamic.array, conversion*)

CALL –ICONVS (*return.array, dynamic.array, conversion*)

CALL !ICONVS (*return.array, dynamic.array, conversion*)

Description

Use the ICONVS function to convert each element of *dynamic.array* to a specified internal storage format.

conversion is an expression that evaluates to one or more valid conversion codes, separated by value marks (ASCII 253).

Each element of *dynamic.array* is converted to the internal format specified by *conversion* and is returned in a dynamic array. If multiple codes are used, they are applied from left to right. The first conversion code converts the value of each element of *dynamic.array*. The second conversion code converts the value of each element of the output of the first conversion, and so on.

If *dynamic.array* evaluates to the null value, null is returned. If an element of *dynamic.array* is the null value, null is returned for that element. If *conversion* evaluates to the null value, the ICONV function fails, the program terminates, and a run-time error message is produced.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

The STATUS function reflects the result of the conversion:

0 The conversion is successful.

1 An element of *dynamic.array* is invalid. An empty string is returned, unless *dynamic.array* is the null value, in which case null is returned.

2 *conversion* is invalid.

3 Successful conversion of possibly invalid data.

See the STATUS function in Volume 2 for more information.

For information about converting elements in a dynamic array to an external format, see the OCONVS function in Volume 2.

IF statement

Syntax

IF *expression* {THEN *statements* [ELSE *statements*] | ELSE *statements*}

IF *expression*

{THEN *statements*

[ELSE *statements*] |

ELSE *statements*}

IF *expression* {THEN
statements

END [ELSE
statements

END] | ELSE
statements

END}

IF *expression*

{THEN
statements

END

[ELSE
statements

END] |

ELSE
statements

END }

Description

Use the IF statement to determine program flow based on the evaluation of *expression*. If the value of *expression* is true, the THEN statements are executed. If the value of *expression* is false, the THEN statements are ignored and the ELSE statements are executed. If *expression* is the null value, *expression* evaluates to false. If no ELSE statements are present, program execution continues with the next executable statement.

The IF statement must contain either a THEN clause or an ELSE clause. It need not include both.

Use the ISNULL function with the IF statement when you want to test whether the value of a variable is the null value. This is the only way to test for the null value since null cannot be equal to any value, including itself. The syntax is:

IF ISNULL (*expression*) ...

You can write IF...THEN statements on a single line or separated onto several lines. Separating statements onto several lines can improve readability. Either way, the statements are executed identically.

You can nest IF...THEN statements. If the THEN or ELSE statements are written on more than one line, you must use an END statement as the last statement of the THEN or ELSE statements.

Conditional Compilation

You can specify the conditions under which all or part of a BASIC program is to be compiled, using a modified version of the IF statement. The syntax of the conditional compilation statement is the same as that of the IF statement except for the test expression, which must be one of the following: $TRUE, $T, $FALSE, or $F.

Example

```
X=10
IF X>5 THEN PRINT 'X IS GREATER THAN 5';Y=3
*
IF Y>5 THEN STOP ELSE Z=9;  PRINT 'Y IS LESS THAN 5'
*
IF Z=9 THEN PRINT 'Z EQUALS 9'
ELSE PRINT 'Z DOES NOT EQUAL 9' ;  STOP
*
IF Z=9 THEN
```

```
   GOTO 10
END ELSE
   STOP
END
*
10*
IF Y>4
   THEN
      PRINT 'Y GREATER THAN 4'
   END
   ELSE
      PRINT 'Y IS LESS THAN 4'
   END
```

This is the program output:

```
X IS GREATER THAN 5
Y IS LESS THAN 5
Z EQUALS 9
Y IS LESS THAN 4
```

Syntax

IFS (*dynamic.array, true.array, false.array*)

CALL –IFS (*return.array, dynamic.array, true.array, false.array*)

CALL !IFS (*return.array, dynamic.array, true.array, false.array*)

Description

Use the IFS function to return a dynamic array whose elements are chosen individually from one of two dynamic arrays based on the contents of a third dynamic array.

Each element of *dynamic.array* is evaluated. If the element evaluates to true, the corresponding element from *true.array* is returned to the same element of a new dynamic array. If the element evaluates to false, the corresponding element from *false.array* is returned. If there is no corresponding element in the correct response array, an empty string is returned for that element. If an element is the null value, that element evaluates to false.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

ILPROMPT function

Syntax

ILPROMPT (*in.line.prompt*)

Description

Use the ILPROMPT function to evaluate a string containing UniVerse in-line prompts.

in.line.prompt is an expression that evaluates to a string containing in-line prompts. In-line prompts have the following syntax:

<<[*control*,] ... *text* [,*option*]>>

control is an option that specifies the characteristics of the prompt. Separate multiple control options with commas. Possible control options are:

A — Always prompts when the sentence containing the control option is executed. If this option is not specified, the input value from a previous execution of this prompt is used.

C*n* — Uses the word in the *n*th position in the command line as the input value. (The verb is in position 1.)

F(*filename*) *record.ID* [,*fm* [,*vm* [,*sm*]]])

Finds input value in *record.ID* in *filename*. Optionally, extract a value (*vm*) or subvalue (*sm*) from the field (*fm*).

I*n* — Uses the word in the *n*th position in the command line as the input value, but prompts if word *n* was not entered.

P — Saves the input from an in-line prompt. BASIC uses the input for all in-line prompts with the same prompt text until the saved input is overwritten by a prompt with the same prompt text and with a control option of A, C, I, or S, or until control returns to the UniVerse prompt. The P option saves the input from an in-line prompt in the current paragraph, or in other paragraphs.

R — Repeats the prompt until **Return** is pressed.

R(*string*) — Repeats the prompt until **Return** is pressed, and inserts *string* between each entry.

S*n*	Takes the *n*th word from the command but uses the most recent command entered at the UniVerse level to execute the paragraph, rather than an argument in the paragraph. Use this option in nested paragraphs.
@(CLR)	Clears the screen.
@(BELL)	Rings the terminal bell.
@(TOF)	Positions the prompt at the top left of the screen.
@(*col*, *row*)	Prompts at this column and row number on the terminal.

text is the prompt text to display. If you want to include quotation marks (single or double) or backslashes as delimiters within the prompt text, you must enclose the entire text string in a set of delimiters different from the delimiters you are using within the text string. For example, to print the following prompt text:

```
'P'RINTER OR 'T'ERMINAL
```

you must specify the prompt text as:

```
\'P'RINTER OR 'T'ERMINAL\
```

or

```
"'P'RINTER OR 'T'ERMINAL"
```

option can be any valid ICONV conversion or matching pattern (see the MATCH operator in Volume 2). A conversion must be in parentheses.

If *in.line.prompt* evaluates to the null value, the ILPROMPT function fails, the program terminates, and a run-time error message appears.

If the in-line prompt has a value, that value is substituted for the prompt. If the in-line prompt does not have a value, the prompt is displayed to request an input value when the sentence is executed. The value entered at the prompt is then substituted for the in-line prompt.

Once a value has been entered for a particular prompt, the prompt will continue to have that value until a CLEARPROMPTS statement is executed, unless the control option A is specified. CLEARPROMPTS clears all values entered for in-line prompts.

You can enclose prompts within prompts.

Example

```
A="This is your number. - <<number>>"
PRINT ILPROMPT(A)
PRINT ILPROMPT("Your number is <<number>>, and your letter is
   <<letter>>.")
```

This is the program output:

```
number=5
This is your number. - 5
letter=K
Your number is 5, and your letter is K.
```

Syntax

INCLUDE [*filename*] *program*

INCLUDE *program* FROM *filename*

Description

Use the INCLUDE statement to direct the compiler to insert the source code in the record *program* and compile it along with the main program. The INCLUDE statement differs from the $CHAIN statement in that the compiler returns to the main program and continues compiling with the statement following the INCLUDE statement.

When *program* is specified without *filename*, *program* must be a record in the same file as the program currently containing the INCLUDE statement.

If *program* is a record in a different file, the name of the file in which it is located must be specified in the INCLUDE statement, followed by the name of the program. The filename must specify a type 1 or type 19 file defined in the VOC file.

You can nest INCLUDE statements.

The INCLUDE statement is a synonym for the $INCLUDE and #INCLUDE statements.

Example

```
PRINT "START"
INCLUDE END
PRINT "FINISH"
```

When this program is compiled, the INCLUDE statement inserts code from the program END (see the example on the END statement page). This is the program output:

```
START
THESE TWO LINES WILL PRINT ONLY
WHEN THE VALUE OF 'A' IS 'YES'.

THIS IS THE END OF THE PROGRAM
FINISH
```

INDEX function

Syntax

INDEX (*string*, *substring*, *occurrence*)

Description

Use the INDEX function to return the starting character position for the specified occurrence of *substring* in *string*.

string is an expression that evaluates to any valid string. *string* is examined for the substring expression.

occurrence specifies which occurrence of *substring* is to be located.

When *substring* is found and if it meets the occurrence criterion, the starting character position of the substring is returned. If *substring* is an empty string, 1 is returned. If the specified occurrence of the substring is not found, or if *string* or *substring* evaluate to the null value, 0 is returned.

If *occurrence* evaluates to the null value, the INDEX function fails, the program terminates, and a run-time error message is produced.

PICK, IN2, and REALITY Flavors

In PICK, IN2, and REALITY flavor accounts, the search continues with the next character regardless of whether it is part of the matched substring. Use the COUNT.OVLP option of the $OPTIONS statement to get this behavior in IDEAL and INFORMATION flavor accounts.

Example

```
Q='AAA11122ABB1619MM'
P=INDEX(Q,1,4)
PRINT "P= ",P
*
X='XX'
Y=2
Q='P1234XXOO1299XX00P'
TEST=INDEX(Q,X,Y)
PRINT "TEST= ",TEST
*
Q=INDEX("1234",'A',1)
PRINT "Q= ",Q
* The substring cannot be found.
```

```
*
POS=INDEX('222','2',4)

PRINT "POS= ",POS
* The occurrence (4) of the substring does not exist.
```

This is the program output:

```
P=       12
TEST=    14
Q=       0
POS=     0
```

INDEXS function

Syntax

INDEXS (*dynamic.array*, *substring*, *occurrence*)

CALL –INDEXS (*return.array*, *dynamic.array*, *substring*, *occurrence*)

CALL !INDEXS (*return.array*, *dynamic.array*, *substring*, *occurrence*)

Description

Use the INDEXS function to return a dynamic array of the starting column positions for a specified occurrence of a substring in each element of *dynamic.array*.

Each element is examined for *substring*.

occurrence specifies which occurrence of *substring* is to be located.

When *substring* is found, and if it meets the occurrence criterion, the starting column position of the substring is returned. If *substring* is an empty string, 1 is returned. If the specified occurrence of *substring* cannot be found, 0 is returned.

If *dynamic.array* evaluates to the null value, 0 is returned. If any element of *dynamic.array* is null, 0 is returned for that element. If *occurrence* is the null value, the INDEXS function fails, the program terminates, and a run-time error message is produced.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

INDICES (*file.variable* [,*indexname*])

Description

Use the INDICES function to return information about the secondary key indices in a file.

file.variable specifies an open file.

indexname is the name of a secondary index in the specified file.

If only *file.variable* is specified, a dynamic array is returned that contains the index names of all secondary indices for the file. The index names are separated by field marks. If *file.variable* has no indices, an empty string is returned.

If *indexname* is specified, information is returned in a dynamic array for *indexname*. Field 1 of the dynamic array contains the following information:

Field 1 of Dynamic Arrays

Value	Value can be	Description
Value 1	D I A S C	Data descriptor index. I-descriptor index. A-descriptor index. S-descriptor index. A- or S-descriptor index with correlative in field 8.
Value 2	1 empty	Index needs rebuilding. Index does not need rebuilding.
Value 3	1 empty	Empty strings are not indexed. Empty strings are indexed.
Value 4	1 empty	Automatic updating enabled. Automatic updating disabled.
Value 5	pathname empty	Full pathname of the index file. File is a distributed file.
Value 6	1 empty	Updates are pending. No updates pending.

Field 1 of Dynamic Arrays (Continued)

Value	Value can be		Description
Value 7	L R		Left-justified. Right-justified.
Value 8	N U		Nonunique. Unique.
Value 9	part numbers		Subvalued list of distributed file part numbers.
Value 10	1 empty	Index needs building No build needed	Subvalued list corresponding to subvalues in Value 9.
Value 11	1 empty	Empty strings not indexed Empty strings indexed	Subvalued list corresponding to subvalues in Value 9.
Value 12	1 empty	Updating enabled Updating disabled	Subvalued list corresponding to subvalues in Value 9.
Value 13	index pathnames		Subvalued list of pathnames for indices on distributed file part files, corresponding to subvalues in Value 9.
Value 14	1 empty	Updates pending No updates pending	Subvalued list corresponding to subvalues in Value 9.
Value 15	L R	Left-justified Right-justified	Subvalued list corresponding to subvalues in Value 9.
Value 16	N U	Nonunique Unique	Subvalued list corresponding to subvalues in Value 9.

If Value 1 of Field 1 is D, A, or S, Field 2 contains the field location (that is, the field number), and Field 6 contains either S (single-valued field) or M (multivalued field).

If Value 1 of Field 1 is I, the other fields of the dynamic array contain the following information, derived from the I-descriptor in the file dictionary:

Field	Value can be
Field 2	I-type expression
Field 3	Output conversion code

Field	Value can be
Field 4	Column heading
Field 5	Width, justification
Field 6	S – single-valued field M – multivalued field
Field 7	Association name
Fields 8–18	Empty
Fields 19–*nn*	Compiled I-descriptor code

If Value 1 of Field 1 is C, the other fields of the dynamic array contain the following information, derived from the A- or S-descriptor in the file dictionary:

Field	Value can be
Field 2	Field number (location of field)
Field 3	Column heading
Field 4	Association code
Fields 5–6	Empty
Field 7	Output conversion code
Field 8	Correlative code
Field 9	L or R (justification)
Field 10	Width of display column

If either *file.variable* or *indexname* is the null value, the INDICES function fails, the program terminates, and a run-time error message is produced.

Any file updates executed in a transaction (that is, between a BEGIN TRANSACTION statement and a COMMIT statement) are not accessible to the INDICES function until after the COMMIT statement has been executed.

INMAT function

Syntax

INMAT ([*array*])

Description

Use the INMAT function to return the number of array elements that have been loaded after the execution of a MATREAD, MATREADL, MATREADU, or MATPARSE statement, or to return the modulo of a file after the execution of an OPEN statement. You can also use the INMAT function after a DIM statement to determine whether the DIM statement failed due to lack of available memory. If a preceding DIM statement fails, INMAT returns a value of 1.

If the matrix assignment exceeds the number of elements specified in its dimensioning statement, the zero element is loaded by the MATREAD, MATREADL, MATREADU, or MATPARSE statement. If the array dimensioning is too small and the zero element has been loaded, the INMAT function returns a value of 0.

If *array* is specified, the INMAT function returns the current dimensions of the array. If *array* is the null value, the INMAT function fails, the program terminates, and a run-time error message is produced.

Example

```
DIM X(6)
D='123456'
MATPARSE X FROM D,''
Y=INMAT()
PRINT 'Y= ':Y
*
DIM X(5)
A='CBDGFH'
MATPARSE X FROM A,''
C=INMAT()
PRINT 'C= ':C
*
OPEN '','VOC' TO FILE ELSE STOP
T=INMAT()
PRINT 'T= ':T
```

This is the program output:

```
Y= 6
C= 0
T= 23
```

INPUT statement

Syntax

INPUT *variable* [,*length*] [:] [_]

INPUT @ (*col*, *row*) [, | :] *variable* [,*length*] [:] [*format*] [_]

INPUTIF @ (*col*, *row*) [, | :] *variable* [,*length*] [:] [*format*] [_]
[THEN *statements*] [ELSE *statements*]

Description

Use the INPUT statement to halt program execution and prompt the user to enter a response. Data entered at the terminal or supplied by a DATA statement in response to an INPUT statement is assigned to *variable*. Input supplied by a DATA statement is echoed to the terminal. If the response is a RETURN with no preceding data, an empty string is assigned to *variable*.

The INPUT statement has two syntaxes. The first syntax displays a prompt and assigns the input to *variable*. The second syntax specifies the location of the input field on the screen and lets you display the current value of *variable*. Both the current value and the displayed input can be formatted.

Use the INPUTIF statement to assign the contents of the type-ahead buffer to a variable. If the type-ahead buffer is empty, the ELSE statements are executed, otherwise any THEN statements are executed.

Use the @ expression to specify the position of the input field. The prompt is displayed one character to the left of the beginning of the field, and the current value of *variable* is displayed as the value in the input field. The user can edit the displayed value or enter a new value. If the first character typed in response to the prompt is an editing key, the user can edit the contents of the field. If the first character typed is anything else, the field's contents are deleted and the user can enter a new value. Editing keys are defined in the *terminfo* files; they can also be defined by the KEYEDIT statement.

col and *row* are expressions that specify the column and row positions of the input prompt. The prompt is positioned one character to the left of the input field. Because the prompt character is positioned to the left of the col position, you must set the prompt to the empty string if you want to use column 0. Otherwise, the screen is erased before the prompt appears.

length specifies the maximum number of characters allowed as input. When the maximum number of characters is entered, input is terminated. If the @ expression is used, the newline is suppressed.

If *length* evaluates to less than 0 (for example, –1), the input buffer is tested for the presence of characters. If characters are present, *variable* is set to 1, otherwise it is set to 0. No input is performed.

If you use the underscore (_) with the length expression, the user must enter the RETURN manually at the terminal when input is complete. Only the specified number of characters is accepted.

Use a format expression to validate input against a format mask and to format the displayed input field. The syntax of the format expression is the same as that for the FMT function. If you specify a length expression together with a format expression, length checking is performed. If input does not conform to the format mask, an error message appears at the bottom of the screen, prompting the user for the correct input.

The colon (:) suppresses the newline after input is terminated. This allows multiple input prompts on a single line.

The default prompt character is a question mark. Use the PROMPT statement to reassign the prompt character.

The INPUT statement prints only the prompt character on the screen. To print a variable name or prompt text along with the prompt, precede the INPUT statement with a PRINT statement.

The INPUT statement lets the user type ahead when entering a response. Users familiar with a sequence of prompts can save time by entering data at their own speed, not waiting for all prompts to be displayed. Responses to a sequence of INPUT prompts are accepted in the order in which they are entered.

If *col*, *row*, *length*, or *format* evaluate to the null value, the INPUT statement fails, the program terminates, and a run-time error message is produced. If *variable* is the null value and the user types the TRAP key, null is retained as the value of *variable*.

PICK Flavor

In a PICK flavor account, the syntax of the INPUT and INPUT @ statements includes THEN and ELSE clauses:

INPUT *variable* [,*length*] [:] [_] [THEN *statements*] [ELSE *statements*]

INPUT @ (*col*, *row*) [, | :] *variable* [,*length*] [:] [*format*] [_]
[THEN *statements*] [ELSE *statements*]

To use THEN and ELSE clauses with INPUT statements in other flavors, use the INPUT.ELSE option of the $OPTIONS statement.

PICK, IN2, and REALITY Flavors

In PICK, IN2, and REALITY flavors, values supplied by a DATA statement are not echoed. To suppress echoing input from DATA statements in IDEAL and INFORMATION flavors, use the SUPP.DATA.ECHO option of the $OPTIONS statement.

Examples

In the following examples of program output, boldface type indicates words the user types. In the first example the value entered is assigned to the variable NAME:

Source Lines	Program Output
`INPUT NAME` `PRINT NAME`	`? `**`Dave`** `Dave`

In the next example the value entered is assigned to the variable CODE. Only the first seven characters are recognized. A RETURN and a LINEFEED automatically occur.

Source Lines	Program Output
`INPUT CODE, 7` `PRINT CODE`	`? `**`1234567`** `1234567`

In the next example the user can enter more than two characters. The program waits for a RETURN to end input, but only the first two characters are assigned to the variable YES.

Source Lines	Program Output
`INPUT YES, 2_` `PRINT YES`	`? `**`1234`** `12`

In the next example the colon inhibits the automatic LINEFEED after the RETURN:

Source Lines	Program Output
`INPUT YES, 2_:` `PRINT "=",YES`	`? HI THERE =HI`

In the next example the input buffer is tested for the presence of characters. If characters are present, VAR is set to 1, otherwise it is set to 0. No input is actually done.

Source Lines	Program Output
`INPUT VAR, -1` `PRINT VAR`	`0`

In the next example the PRINT statement puts INPUT NAME before the input prompt:

Source Lines	Program Output

`PRINT "INPUT NAME":`
`INPUT NAME`
`PRINT NAME`
	`INPUT NAME? Dave`
	`Dave`

In the next example the contents of X are displayed at column 5, row 5 in a field of 10 characters. The user edits the field, replacing its original contents (CURRENT) with new contents (NEW). The new input is displayed. If the PRINT statement after the INPUT statement were not used, X would be printed immediately following the input field on the same line, since INPUT with the @ expression does not execute a LINEFEED after a RETURN.

Source Lines	Program Output

`PRINT @(-1)`
`X = "CURRENT"`
`INPUT @(5,5) X,10`
`PRINT`
`PRINT X`
	`?NEW_____`
	`NEW`

INPUTCLEAR statement

Syntax

INPUTCLEAR

Description

Use the INPUTCLEAR statement to clear the type-ahead buffer. You can use this statement before input prompts so input is not affected by unwanted characters.

Example

```
PRINT "DO YOU WANT TO CONTINUE (Y/N)?"
INPUTCLEAR
INPUT ANSWER, 1
```

Syntax

INPUTDISP [@(*col*, *row*) [, | :]] *variable* [*format*]

Description

Use the INPUTDISP statement with an @ expression to position the cursor at a specified location and define a format for the variable to print. The current contents of *variable* are displayed as the value in the defined field.

col specifies the column position, and *row* specifies the row position.

format is an expression that defines how the variable is to be displayed in the output field. The syntax of the format expression is the same as that for the FMT function.

Example

```
PRINT @(-1)
X = "CURRENT LINE"
INPUTDISP @(5,5),X"10T"
```

The program output on a cleared screen is:

```
CURRENT
LINE
```

INPUTERR statement

Syntax

INPUTERR [*error.message*]

Description

Use the INPUTERR statement to print a formatted error message on the bottom line of the terminal. *error.message* is an expression that evaluates to the error message text. The message is cleared by the next INPUT statement or is overwritten by the next INPUTERR or PRINTERR statement. INPUTERR clears the type-ahead buffer.

error.message can be any BASIC expression. The elements of the expression can be numeric or character strings, variables, constants, or literal strings. The null value cannot be output. The expression can be a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no error message is designated, a blank line is printed. If *error.message* evaluates to the null value, the default error message is printed:

```
Message ID is NULL:  undefined error
```

Expressions separated by commas are printed at preset tab positions. The default tabstop setting is 10 characters. For information about changing the default setting, see the TABSTOP statement in Volume 2. Multiple commas can be used together to cause multiple tabulations between expressions.

Expressions separated by colons are concatenated: that is, the expression following the colon is printed immediately after the expression preceding the colon.

See also the PRINTERR statement in Volume 2.

INPUTIF statement

Use the INPUTIF statement to assign the contents of the type-ahead buffer to a variable. For details, see the INPUT statement.

INPUTNULL statement

Syntax

INPUTNULL *character*

Description

Use the INPUTNULL statement to define a character to be recognized as an empty string when it is input in response to an INPUT statement. If the only input to the INPUT statement is *character*, that character is recognized as an empty string. *character* replaces the default value of the INPUT variable with an empty string. If *character* evaluates to the null value, the INPUTNULL statement fails, the program terminates, and a run-time error message is produced.

You can also assign an empty string to the variable used in the INPUT @ statement before executing the INPUT @. In this case entering a RETURN leaves the variable set to the empty string.

Note: Although the name of this statement is INPUTNULL, it does not define *character* to be recognized as the null value. It defines it to be recognized as an empty string.

Syntax

INPUTTRAP [*trap.chars*] {GOTO | GOSUB} *label* [,*label* ...]

Description

Use the INPUTTRAP statement to branch to a program label or subroutine when a trap character is input. Execution is passed to the statement label which corresponds to the trap number of the trap character. If the trap number is larger than the number of labels, execution is passed to the statement specified by the last label in the list.

trap.chars is an expression that evaluates to a string of characters, each of which defines a trap character. The first character in the string is defined as trap one. Additional characters are assigned consecutive trap numbers. Each trap character corresponds to one of the labels in the label list. If *trap.chars* evaluates to the null value, the INPUTTRAP statement fails, the program terminates, and a run-time error message is produced.

Using GOTO causes execution to be passed to the specified statement label. Control is not returned to the INPUTTRAP statement except by the use of another trap. Using GOSUB causes execution to be passed to the specified subroutine, but control can be returned to the INPUTTRAP statement by a RETURN statement. Note that control is returned to the statement following the INPUTTRAP statement, not the INPUT @ statement that received the trap.

INS statement

Syntax

INS *expression* BEFORE *dynamic.array* < *field#* [,*value#* [,*subvalue#*]] >

Description

Use the INS statement to insert a new field, value, or subvalue into the specified *dynamic.array*.

expression specifies the value of the new element to be inserted.

dynamic.array is an expression that evaluates to the dynamic array to be modified.

field#, *value#*, and *subvalue#* specify the type and position of the new element to be inserted and are called delimiter expressions.

There are three possible outcomes of the INS statement, depending on the delimiter expressions specified.

Case 1: If both *value#* and *subvalue#* are omitted or are 0, INS inserts a new field with the value of *expression* into the dynamic array.

- If *field#* is positive and less than or equal to the number of fields in *dynamic.array*, the value of *expression* followed by a field mark is inserted before the field specified by *field#*.
- If *field#* is –1, a field mark followed by the value of *expression* is appended to the last field in *dynamic.array*.
- If *field#* is positive and greater than the number of fields in *dynamic.array*, the proper number of field marks followed by the value of *expression* are appended so that the value of *field#* is the number of the new field.

Case 2: If *value#* is nonzero and *subvalue#* is omitted or is 0, INS inserts a new value with the value of *expression* into the dynamic array.

- If *value#* is positive and less than or equal to the number of values in the field, the value of *expression* followed by a value mark is inserted before the value specified by *value#*.
- If *value#* is –1, a value mark followed by the value of *expression* is appended to the last value in the field.

- If *value#* is positive and greater than the number of values in the field, the proper number of value marks followed by the value of *expression* are appended to the last value in the specified field so that the number of the new value in the field is *value#*.

Case 3: If *field#*, *value#*, and *subvalue#* are all specified, INS inserts a new subvalue with the value of *expression* into the dynamic array.

- If *subvalue#* is positive and less than or equal to the number of subvalues in the value, the value of *expression* following by a subvalue mark is inserted before the subvalue specified by *subvalue#*.
- If *subvalue#* is –1, a subvalue mark followed by *expression* is appended to the last subvalue in the value.
- If *subvalue#* is positive and greater than the number of subvalues in the value, the proper number of subvalue marks followed by the value of *expression* are appended to the last subvalue in the specified value so that the number of the new subvalue in the value is *subvalue#*.

If all delimiter expressions are 0, the original string is returned.

In IDEAL, PICK, PIOPEN, and REALITY flavor accounts, if *expression* is an empty string and the new element is appended to the end of the dynamic array, the end of a field, or the end of a value, the dynamic array, field, or value is left unchanged. Additional delimiters are not appended. Use the EXTRA.DELIM option of the $OPTIONS statements to make the INS statement append a delimiter to the dynamic array, field, or value.

If *expression* evaluates to the null value, null is inserted into *dynamic.array*. If *dynamic.array* evaluates to the null value, it remains unchanged by the insertion. If the INS statement references a subelement of an element whose value is the null value, the dynamic array is unchanged.

If any delimiter expression is the null value, the INS statement fails, the program terminates, and a run-time error message is produced.

INFORMATION and IN2 Flavors

In INFORMATION and IN2 flavor accounts, if *expression* is an empty string and the new element is appended to the end of the dynamic array, the end of a field, or the

end of a value, a delimiter is appended to the dynamic array, field, or value. Use the –EXTRA.DELIM option of the $OPTIONS statement to make the INS statement work as it does in IDEAL, PICK, and REALITY flavor accounts.

Examples

In the following examples a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

The first example inserts the character # before the first field and sets Q to #FFF1V2V3S6F9F5F7V3:

```
R=@FM:@FM:1:@VM:2:@VM:3:@SM:6:@FM:9:@FM:5:@FM:7:@VM:3
Q=R
INS "#" BEFORE Q<1,0,0>
```

The next example inserts a # before the third value of field 3 and sets the value of Q to FF1V2V#V3S6F9F5F7V3:

```
Q=R
INS "#" BEFORE Q<3,3,0>
```

The next example inserts a value mark followed by a # after the last value in the field and sets Q to FF1V2V3S6F9V#F5F7V3:

```
Q=R
INS "#" BEFORE Q<4,-1,0>
```

The next example inserts a # before the second subvalue of the second value of field 3 and sets Q to FF1V2S#V3S6F9F5F7V3:

```
Q=R
INS "#" BEFORE Q<3,2,2>
```

Syntax

INSERT (*dynamic.array, field#, value#, subvalue#, expression*)

INSERT (*dynamic.array, field#* [,*value#* [,*subvalue#*]] ; *expression*)

Description

Use the INSERT function to return a dynamic array that has a new field, value, or subvalue inserted into the specified dynamic array.

dynamic.array is an expression that evaluates to a dynamic array.

field#, *value#*, and *subvalue#* specify the type and position of the new element to be inserted and are called delimiter expressions. *value#* and *subvalue#* are optional, but if either is omitted, a semicolon (;) must precede *expression*, as shown in the second syntax line.

expression specifies the value of the new element to be inserted.

There are three possible outcomes of the INSERT function, depending on the delimiter expressions specified.

Case 1: If both *value#* and *subvalue#* are omitted or are 0, INSERT inserts a new field with the value of *expression* into the dynamic array.

- If *field#* is positive and less than or equal to the number of fields in *dynamic.array*, the value of *expression* followed by a field mark is inserted before the field specified by *field#*.
- If *field#* is –1, a field mark followed by the value of *expression* is appended to the last field in *dynamic.array*.
- If *field#* is positive and greater than the number of fields in *dynamic.array*, the proper number of field marks followed by the value of *expression* are appended so that the value of *field#* is the number of the new field.

Case 2: If *value#* is nonzero and *subvalue#* is omitted or is 0, INSERT inserts a new value with the value of *expression* into the dynamic array.

- If *value#* is positive and less than or equal to the number of values in the field, the value of *expression* followed by a value mark is inserted before the value specified by *value#*.

- If *value#* is –1, a value mark followed by the value of *expression* is appended to the last value in the field.
- If *value#* is positive and greater than the number of values in the field, the proper number of value marks followed by the value of *expression* are appended to the last value in the specified field so that the number of the new value in the field is *value#*.

Case 3: If *field#*, *value#*, and *subvalue#* are all specified, INSERT inserts a new subvalue with the value of *expression* into the dynamic array.

- If *subvalue#* is positive and less than or equal to the number of subvalues in the value, the value of *expression* following by a subvalue mark is inserted before the subvalue specified by *subvalue#*.
- If *subvalue#* is –1, a subvalue mark followed by *expression* is appended to the last subvalue in the value.
- If *subvalue#* is positive and greater than the number of subvalues in the value, the proper number of subvalue marks followed by the value of *expression* are appended to the last subvalue in the specified value so that the number of the new subvalue in the value is *subvalue#*.

In IDEAL, PICK, PIOPEN, and REALITY accounts, if *expression* is an empty string and the new element is appended to the end of the dynamic array, the end of a field, or the end of a value, the dynamic array, field, or value is left unchanged. Additional delimiters are not appended. Use the EXTRA.DELIM option of the $OPTIONS statement to make the INSERT function append a delimiter to the dynamic array, field, or value.

If *expression* evaluates to the null value, null is inserted into *dynamic.array*. If *dynamic.array* evaluates to the null value, it remains unchanged by the insertion. If any delimiter expression is the null value, the INSERT function fails, the program terminates, and a run-time error message is produced.

INFORMATION and IN2 Flavors

In INFORMATION and IN2 flavor accounts, if *expression* is an empty string and the new element is appended to the end of the dynamic array, the end of a field, or the end of a value, a delimiter is appended to the dynamic array, field, or value. Use

the –EXTRA.DELIM option of the $OPTIONS statement to make the INSERT function work as it does in IDEAL, PICK, and REALITY flavor accounts.

Examples

In the following examples a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

The first example inserts the character # before the first field and sets Q to #FFF1V2V3S6F9F5F7V:

```
R=@FM:@FM:1:@VM:2:@VM:3:@SM:6:@FM:9:@FM:5:@FM:7:@VM:3
Q=INSERT(R,1,0,0,"#")
```

The next example inserts a # before the third value of field 3 and sets the value of Q to FF1V2V#V3S6F9F5F7V3:

```
Q=INSERT(R,3,3;"#")
```

The next example inserts a value mark followed by a # after the last value in the field and sets Q to FF1V2V3S6F9V#F5F7V3:

```
Q=INSERT(R,4,-1,0,"#")
```

The next example inserts a # before the second subvalue of the second value of field 3 and sets Q to FF1V2S#V3S6F9F5F7V3:

```
Q=INSERT(R,3,2,2;"#")
```

INT function

Syntax

INT (*expression*)

Description

Use the INT function to return the integer portion of an expression.

expression must evaluate to a numeric value. Any arithmetic operations specified are calculated using the full accuracy of the system. The fractional portion of the value is truncated, not rounded, and the integer portion remaining is returned.

If *expression* evaluates to the null value, null is returned.

Example

```
PRINT "123.45 ", INT(123.45)
PRINT "454.95 ", INT(454.95)
```

This is the program output:

```
123.45   123
454.95   454
```

Syntax

ISNULL (*variable*)

Description

Use the ISNULL function to test whether a variable is the null value. If *variable* is the null value, 1 (true) is returned, otherwise 0 (false) is returned. This is the only way to test for the null value since the null value is not equal to any value, including itself.

Example

```
X = @NULL
Y = @NULL.STR
PRINT ISNULL(X), ISNULL(Y)
```

This is the program output:

```
1  0
```

ISNULLS function

Syntax

ISNULLS (*dynamic.array*)

CALL –ISNULLS (*return.array, dynamic.array*)

Description

Use the ISNULLS function to test whether any element of *dynamic.array* is the null value. A dynamic array is returned, each of whose elements is either 1 (true) or 0 (false). If an element in *dynamic.array* is the null value, 1 is returned, otherwise 0 is returned. This is the only way to test for the null value since the null value is not equal to any value, including itself.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

Example

```
DA = ""
FOR I = 1 TO 7
   DA := I:@FM
   IF I = 5 THEN DA := @NULL.STR:@FM
NEXT I
PRINT ISNULLS(DA)
```

This is the program output:

```
0F0F0F0F0F1F0F0F0
```

Syntax

ITYPE (*i.type*)

Description

Use the ITYPE function to return the value resulting from the evaluation of an I-descriptor expression in a UniVerse file dictionary.

i.type is an expression evaluating to the contents of the compiled I-descriptor. The I-descriptor must have been compiled before the ITYPE function uses it, otherwise you get a run-time error message.

i.type can be set to the I-descriptor to be evaluated in several ways. One way is to read the I-descriptor from a file dictionary into a variable, then use the variable as the argument to the ITYPE function. If the I-descriptor references a record ID, the current value of the system variable @ID is used. If the I-descriptor references field values in a data record, the data is taken from the current value of the system variable @RECORD.

To assign field values to @RECORD, read a record from the data file into @RECORD before invoking the ITYPE function.

If *i.type* evaluates to the null value, the ITYPE function fails, the program terminates, and a run-time error message is produced.

Example

This is the SUN.MEMBER file contents:

```
AW
F1: ACCOUNTING
TRX
F1: MARKETING
JXA
F1: SALES
```

This is the DICT.ITME contents:

```
DEPARTMENT
F1:D
2:1
3:
4:
```

```
5:10L
6:L
```

This is the program source code:

```
OPEN 'SUN.MEMBER' TO FILE ELSE STOP
OPEN 'DICT','SUN.MEMBER' TO D.FILE ELSE STOP
*
READ ITEM.ITYPE FROM D.FILE, 'DEPARTMENT' ELSE STOP
*
EXECUTE 'SELECT SUN.MEMBER'
LOOP
READNEXT @ID DO
*
   READ @FRECORD FROM FILE, @ID THEN
   *
   PRINT @ID: "WORKS IN DEPARTMENT" ITYPE(ITEM.ITYPE)
   END
REPEAT
STOP
END
```

This is the program output:

```
3 records selected to Select List #0
FAW WORKS IN DEPARTMENT ACCOUNTING
TRX WORKS IN DEPARTMENT MARKETING
JXA WORKS IN DEPARTMENT SALES
```

Syntax

KEYEDIT (*function*, *key*) [, (*function*, *key*) ...]

Description

Use the KEYEDIT statement to assign specific editing functions to the keyboard keys for use with the @ expression of an INPUT statement.

key is an expression that specifies the key to be defined as follows:

Decimal Value	ASCII Character
1 through 31	Control keys (ASCII 1 to 31).
32 through 159	Escape keys (**Esc** followed by ASCII 0 to 127).
160 through 287	Function keys (FUNCT followed by ASCII 0 to 127, followed by a RETURN. The FUNCT key is defined by specifying *function*.)

To define the keys, you must know the ASCII codes generated by the keyboard on the terminal you use to edit an INPUT statement field.

function is an expression that assigns an editing function to a key. The codes and their functions are:

Code	Function
1	FUNC key
2	<— (left arrow)
3	RETURN
4	Back space
5	Escape
6	—> (right arrow)
7	Insert character
8	Delete character
9	Insert mode on
10	Insert mode off

Code	Function
11	Clear
12	Erase
13	Insert mode toggle

If either *function* or *key* evaluates to the null value, the KEYEDIT statement fails, the program terminates, and a run-time error message is produced.

Examples

The following example sets up the key expressions for a Viewpoint terminal, on which the following ASCII codes are generated:

Function	Decimal Value	Key
FUNC	27	**Esc**
<—	21	**Ctrl-U**
RETURN	13	**Ctrl-M**
Backspace	8	**Ctrl-H**
—>	6	**Ctrl-F**
Insert CHAR	209	**Ctrl-Return**

The KEYTRAP statement sets up **Ctrl-B** as a trapped key. If the STATUS function returns 1, it means trap key 1 (**Ctrl-B**) was entered.

```
KEYEDIT (1,27), (2,21), (3,13), (4,8), (6,6), (7,209)
KEYTRAP (1,2)
1 PRINT @(-1)
X = "CURRENT"
INPUT @(5,5),X,10
PRINT
PRINT "X= ":X
PRINT "STATUS() = ":STATUS()
IF STATUS() = 1 THEN STOP
INPUT A
GOTO 1
```

Syntax

KEYEXIT (*value*, *key*) [, (*value*, *key*) ...]

Description

Use the KEYEXIT statement to specify exit traps for the keys assigned specific functions by the KEYEDIT statement. When an exit trap key is typed, the variable being edited with the INPUT @ statement remains in its last edited state. To exit the INPUT @ statement restoring the variable to its initial state, use the KEYTRAP statement.

key is an expression that specifies the key to be associated with each exit trap as follows:

Decimal Value	ASCII Character
1 through 31	Control keys (ASCII 1 to 31).
32 through 159	Escape keys (**Esc** followed by ASCII 0 to 127).
160 through 287	Function keys (FUNCT followed by ASCII 0 to 127, followed by a RETURN. The FUNCT key is defined by specifying *function*.)

value is an expression that specifies a user-defined trap number for each key assigned by the KEYEDIT statement.

If either *value* or *key* evaluates to the null value, the KEYEXIT statement fails, the program terminates, and a run-time error message is produced.

KEYEXIT sets the STATUS function to the trap number of any trap key typed by the user.

Examples

The following example sets up **Ctrl-B** as an exit trap key. The STATUS function is set to 1.

```
KEYEXIT (1,2)
```

The next example sets up **Ctrl-K** as an exit trap key. The STATUS function is set to 2.

```
KEYEXIT (2,11)
```

KEYIN function

Syntax

KEYIN ( )

Description

Use the KEYIN function to read a single character from the input buffer and return it. All UniVerse special character handling (such as case inversion, erase, kill, and so on) is disabled. UNIX special character handling (processing of interrupts, XON / XOFF, conversion of CR to LF, and so on) still takes place.

No arguments are required with the KEYIN function; however, parentheses are required.

Syntax

KEYTRAP (*value*, *key*) [, (*value*, *key*) ...]

Description

Use the KEYTRAP statement to specify traps for the keys assigned specific functions by the KEYEDIT statement. When a trap key is typed, the variable being edited with the INPUT @ statement is restored to its initial value. To exit the INPUT @ statement without restoring the variable, you can use the KEYEXIT statement.

key is an expression that specifies the key on the keyboard that is to be associated with each trap as follows:

Decimal Value	ASCII Character
1 through 31	Control keys (ASCII 1 to 31).
32 through 159	Escape keys (**Esc** followed by ASCII 0 to 127).
160 through 287	Function keys (FUNCT followed by ASCII 0 to 127, followed by a RETURN. The FUNCT key is defined by specifying *function*.)

value is an expression that specifies a user-defined trap number for each key assigned by the KEYEDIT statement.

If either *value* or *key* evaluates to the null value, the KEYTRAP statement fails, the program terminates, and a run-time error message is produced.

KEYTRAP sets the STATUS function to the trap number of any trap key typed by the user.

Examples

The following example sets up **Ctrl-B** as a trapped key. The STATUS function is set to 1.

```
KEYTRAP (1,2)
```

The next example defines function key values for the keys **F1**, **F2**, **F3**, and **F4** on a WYSE 50 terminal:

```
KEYEDIT (1,1)
KEYTRAP (1,224),(2,225),(3,226),(4,227)
```

```
PRINT @(-1)
VALUE = "KEY"
INPUT @ (10,10):VALUE
X=STATUS()
BEGIN CASE
   CASE X = 1
    PRINT "FUNCTION KEY 1"
   CASE X =2
    PRINT "FUNCTION KEY 2"
   CASE X =3
    PRINT "FUNCTION KEY 3"
   CASE X =4
    PRINT "FUNCTION KEY 4"
END CASE
PRINT VALUE
STOP
END
```

Syntax

LEFT (*string*, *n*)

Description

Use the LEFT function to extract a substring comprising the first *n* characters of a string, without specifying the starting character position. It is equivalent to the following substring extraction operation:

string [1, *length*]

If *string* evaluates to the null value, null is returned. If *n* evaluates to the null value, the LEFT function fails, the program terminates, and a run-time error message is produced.

Example

```
PRINT LEFT("ABCDEFGH",3)
```

This is the program output:

```
ABC
```

LEN function

Syntax

LEN (*string*)

Description

Use the LEN function to return the number of characters in *string*.

string must be a string value. The characters in *string* are counted, and the count is returned.

The LEN function includes all blank spaces, including trailing blanks, in the calculation.

If *string* evaluates to the null value, 0 is returned.

Example

```
P="PORTLAND, OREGON"
PRINT "LEN(P)= ",LEN(P)
*
NUMBER=123456789
PRINT "LENGTH OF NUMBER IS ",LEN(NUMBER)
```

This is the program output:

```
LEN(P)=  16
LENGTH OF NUMBER IS          9
```

Syntax

LENS (*dynamic.array*)

CALL –LENS (*return.array, dynamic.array*)

CALL !LENS (*return.array, dynamic.array*)

Description

Use the LENS function to return a dynamic array of the number of characters in each element of *dynamic.array.*

Each element of *dynamic.array* must be a string value. The characters in each element of *dynamic.array* are counted, and the counts are returned.

The LENS function includes all blank spaces, including trailing blanks, in the calculation.

If *dynamic.array* evaluates to the null value, 0 is returned. If any element of *dynamic.array* is null, 0 is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

LES function

Syntax

LES (*array1*, *array2*)

CALL –LES (*return.array*, *array1*, *array2*)

CALL !LES (*return.array*, *array1*, *array2*)

Description

Use the LES function to test if elements of one dynamic array are less than or equal to the elements of another dynamic array.

Each element of *array1* is compared with the corresponding element of *array2*. If the element from *array1* is less than or equal to the element from *array2*, a 1 is returned in the corresponding element of a new dynamic array. If the element from *array1* is greater than the element from *array2*, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, the undefined element is evaluated as empty, and the comparison continues.

If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

[LET] *variable* = *expression*

Description

Use the LET statement to assign the value of *expression* to *variable*. See also "assignment statements" for more information about assigning values to variables.

Example

```
LET A=55
LET B=45
LET C=A+B
LET D="55+45="
LET E=D:C
PRINT E
```

This is the program output:

```
55+45=100
```

LN function

Syntax

LN (*expression*)

Description

Use the LN function to calculate the natural logarithm of the value of an expression, using base "e". The value of "e" is approximately 2.71828. *expression* must evaluate to a numeric value greater than 0.

If *expression* is 0 or negative, 0 is returned and a warning is printed. If *expression* evaluates to the null value, null is returned.

Example

```
PRINT LN(6)
```

This is the program output:

```
1.7918
```

Syntax (IDEAL, REALITY)

LOCATE *expression* IN *dynamic.array* [< *field#* [,*value#*] >] [,*start*] [BY *seq*]
SETTING *variable*
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Syntax (PICK)

LOCATE (*expression*, *dynamic.array* [,*field#* [,*value#*]] ; *variable* [;*seq*])
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Syntax (INFORMATION)

LOCATE *expression* IN *dynamic.array* < *field#* [,*value#* [,*subvalue#*]] >
[BY *seq*] SETTING *variable*
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use a LOCATE statement to search *dynamic.array* for *expression* and to return a value indicating one of the following:

- Where *expression* was found in *dynamic.array*
- Where *expression* should be inserted in *dynamic.array* if it was not found

The search can start anywhere in *dynamic.array.*

expression evaluates to the string to be searched for in *dynamic.array.* If *expression* or *dynamic.array* evaluate to the null value, *variable* is set to 0 and the ELSE statements are executed. If *expression* and *dynamic.array* both evaluate to empty strings, *variable* is set to 1 and the THEN statements are executed.

field#, *value#*, and *subvalue#* are delimiter expressions, specifying:

- Where the search is to start in *dynamic.array*
- What kind of element is being searched for

start evaluates to a number specifying the field, value, or subvalue from which to start the search.

In the IDEAL and PICK syntaxes the delimiter expressions specify the level of the search, and *start* specifies the starting position of the search. In INFORMATION syntax the delimiter expressions specify the starting position of the search.

If any delimiter expression or *start* evaluates to the null value, the LOCATE statement fails, the program terminates, and a run-time error message is produced.

variable stores the index of *expression*. *variable* returns a field number, value number, or a subvalue number, depending on the delimiter expressions used. *variable* is set to a number representing one of the following:

- The index of the element containing *expression*, if such an element is found
- An index that can be used in an INSERT function to create a new element with the value specified by *expression*

During the search, fields are processed as single-valued fields even if they contain value or subvalue marks. Values are processed as single values, even if they contain subvalue marks.

The search stops when one of the following conditions is met:

- A field containing *expression* is found.
- The end of the dynamic array is reached.
- A field that is higher or lower, as specified by *seq*, is found.

If the elements to be searched are sorted in one of the ascending or descending ASCII sequences listed below, you can use the BY *seq* expression to end the search. The search ends at the place where *expression* should be inserted to maintain the ASCII sequence, rather than at the end of the list of specified elements.

Use the following values for *seq* to describe the ASCII sequence being searched:

“AL” or “A”	Ascending, left-justified (standard alphanumeric sort)
“AR”	Ascending, right-justified
“DL” or “D”	Descending, left-justified (standard alphanumeric sort)
“DR”	Descending, right-justified

seq does not reorder the elements in *dynamic.array*; it specifies the terminating conditions for the search. If a *seq* expression is used and the elements are not in the sequence indicated by *seq*, an element with the value of *expression* may not be found. If *seq* evaluates to the null value, the statement fails and the program terminates.

The ELSE statements are executed if *expression* is not found. The format of the ELSE statement is the same as that used in the IF...THEN statement.

Use the INFO.LOCATE option of the $OPTIONS statement to use the INFORMATION syntax in other flavors.

IDEAL, REALITY, and PICK Syntax

The following sections describe the three possible outcomes from the different uses of delimiter expressions in the IDEAL, REALITY, and PICK versions of the LOCATE statement.

Case 1: If *field#* and *value#* are omitted, the search starts at the first field in *dynamic.array*.

Case 2: If only *field#* is specified and it is greater than 0, the search starts at the first value in the field indicated by *field#*. If *field#* is less than or equal to 0, both *field#* and *value#* are ignored.

Case 3: If both *field#* and *value#* are specified, the search starts at the first subvalue in the value specified by *value#*, in the field specified by *field#*. If *field#* is greater than 0, but *value#* is less than or equal to 0, LOCATE behaves as though only *field#* is specified.

If a field, value, or subvalue containing *expression* is found, *variable* returns the index of the located field, value, or subvalue relative to the start of *dynamic.array*, *field#*, or *value#*, respectively (not the start of the search). If a field, value, or subvalue containing *expression* is not found, *variable* is set to the number of fields, values, or subvalues in the array plus 1, and the ELSE statements are executed.

INFORMATION Syntax

When you use the INFORMATION flavor syntax of LOCATE, three outcomes can result depending on how the delimiter expressions are used. The results are described as case 1, case 2, and case 3.

Case 1: If both *value#* and *subvalue#* are omitted or are both less than or equal to 0, the search starts at the field indicated by *field#*.

Case 2: If *subvalue#* is omitted or is less than or equal to 0, the search starts at the value indicated by *value#*, in the field indicated by *field#*. If *field#* is less than or equal to 0, *field#* defaults to 1.

Case 3: If *field#*, *value#*, and *subvalue#* are all specified and are all nonzero, the search starts at the subvalue indicated by *subvalue#*, in the value specified by *value#*, in the field specified by *field#*. If *field#* or *value#* are less than or equal to 0, they default to 1.

If a field, value, or subvalue containing *expression* is found, *variable* is set to the index of the located field relative to the start of *dynamic.array*, the field, or the value, respectively (not the start of the search).

If no field containing *expression* is found, *variable* is set to the number of the field at which the search terminated, and the ELSE statements are executed. If no value or subvalue containing *expression* is found, *variable* is set to the number of values or subvalues plus 1, and the ELSE statements are executed.

If *field#*, *value#*, or *subvalue#* is greater than the number of fields in *dynamic.array*, *variable* is set to the value of *field#*, *value#*, or *subvalue#*, respectively, and the ELSE statements are executed.

Examples

The examples show the IDEAL and REALITY flavor LOCATE statement. A field mark is shown by **F**, a value mark is shown by **V**, and a subvalue mark is shown by **S**.

```
Q='X':@SM:"$":@SM:'Y':@VM:'Z':@SM:4:@SM:2:@VM:'B':@VM
PRINT "Q= ":Q

LOCATE "$" IN Q <1> SETTING WHERE ELSE PRINT 'ERROR'
PRINT "WHERE= ",WHERE

LOCATE "$" IN Q <1,1> SETTING HERE ELSE PRINT 'ERROR'
PRINT "HERE= ", HERE

NUMBERS=122:@FM:123:@FM:126:@FM:130:@FM
PRINT "BEFORE INSERT, NUMBERS= ",NUMBERS
NUM= 128
LOCATE NUM IN NUMBERS <2> BY "AR" SETTING X ELSE
NUMBERS = INSERT(NUMBERS,X,0,0,NUM)
PRINT "AFTER INSERT, NUMBERS= ",NUMBERS
END
```

This is the program output:

```
Q= Xs$sYvZs4s2vBv

ERROR
WHERE= 5

HERE=  2

BEFORE INSERT, NUMBERS= 122F123F126F130F
AFTER INSERT, NUMBERS=  122F128F123F126F130F
```

LOCK statement

Syntax

LOCK *expression* [THEN *statements*] [ELSE *statements*]

Description

Use the LOCK statement to protect specified user-defined resources or events against unauthorized use or simultaneous data file access by different users.

There are 64 public semaphore locks in the UniVerse system. They are task synchronization tools but have no intrinsic definitions. You must define the resource or event associated with each semaphore, ensuring that there are no conflicts in definition or usage of these semaphores throughout the entire system.

expression evaluates to a number in the range of 0 through 63 that specifies the lock to be set. A program can reset a lock any number of times and with any frequency desired. If *expression* evaluates to the null value, the LOCK statement fails, the program terminates, and a run-time error message is produced.

If program B tries to set a lock already set by program A, execution of program B is suspended until the first lock is released by program A; execution of program B then continues.

The ELSE clause provides an alternative to this procedure. When a LOCK statement specifies a lock that has already been set, the ELSE clause is executed rather than program execution being suspended.

Program termination does not automatically release locks set in the program. Each LOCK statement must have a corresponding UNLOCK statement. If a program locks the same semaphore more than once during its execution, a single UNLOCK statement releases that semaphore.

The UNLOCK statement can specify the expression used in the LOCK statement to be released. If no expression is used in the UNLOCK statement, all locks set by the program are released.

Alternatively, locks can be released by logging off the system or by executing either the QUIT command or the CLEAR.LOCKS command.

You can check the status of locks with the LIST.LOCKS command; this lists the locks on the screen. The unlocked state is indicated by 0. The locked state is indicated by a number other than 0 (including both positive and negative numbers). The number is the unique signature of the user who has set the lock.

Note: The LOCK statement protects user-defined resources only. The READL, READU, READVL, READVU, MATREADL, and MATREADU statements use a different method of protecting files and records.

Example

The following example sets lock 60, executes the LIST.LOCKS command, then unlocks all locks set by the program:

```
LOCK 60 ELSE PRINT "ALREADY LOCKED"
EXECUTE "LIST.LOCKS"
UNLOCK
```

The program displays the LIST.LOCKS report. Lock 60 is set by user 4.

```
 0:--  1:--  2:--  3:--  4:--  5:--  6:--  7:--
 8:--  9:-- 10:-- 11:-- 12:-- 13:-- 14:-- 15:--
16:-- 17:-- 18:-- 19:-- 20:-- 21:-- 22:-- 23:--
24:-- 25:-- 26:-- 27:-- 28:-- 29:-- 30:-- 31:--
32:-- 33:-- 34:-- 35:-- 36:-- 37:-- 38:-- 39:--
40:-- 41:-- 42:-- 43:-- 44:-- 45:-- 46:-- 47:--
48:-- 49:-- 50:-- 51:-- 52:-- 53:-- 54:-- 55:--
56:-- 57:-- 58:-- 59:-- 60:4  61:-- 62:-- 63:--
```

LOOP statement

Syntax

```
LOOP
        [loop.statements]
        [CONTINUE]

[{WHILE | UNTIL} expression [DO] ]
        [loop.statements]
        [CONTINUE]

REPEAT
```

Description

Use the LOOP statement to start a LOOP...REPEAT program loop. A program loop is a series of statements that executes for a specified number of repetitions or until specified conditions are met.

Use the WHILE clause to indicate that the loop should execute repeatedly as long as the WHILE expression evaluates to true (1). When the WHILE expression evaluates to false (0), repetition of the loop stops, and program execution continues with the statement following the REPEAT statement.

Use the UNTIL clause to put opposite conditions on the LOOP statement. The UNTIL clause indicates that the loop should execute repeatedly as long as the UNTIL expression evaluates to false (0). When the UNTIL expression evaluates to true (1), repetition of the loop stops, and program execution continues with the statement following the REPEAT statement.

If a WHILE or UNTIL expression evaluates to the null value, the condition is false.

expression can also contain a conditional statement. Any statement that takes a THEN or an ELSE clause can be used as *expression*, but without the THEN or ELSE clause. When the conditional statement would execute the ELSE clause, *expression* evaluates to false; when the conditional statement would execute the THEN clause, *expression* evaluates to true. A LOCKED clause is not supported in this context.

You can use multiple WHILE and UNTIL clauses in a LOOP...REPEAT loop. You can also nest LOOP...REPEAT loops. If a REPEAT statement is encountered without a previous LOOP statement, an error occurs during compilation.

You can use the CONTINUE statement within LOOP...REPEAT to transfer control to the next iteration of the loop from any point in the loop.

Although it is possible to exit the loop by means other than the conditional WHILE and UNTIL statements (for example, by using GOTO or GOSUB in the DO statements), it is not recommended. Such a programming technique is not in keeping with good structured programming practice.

Examples

Source Lines	Program Output

`X=0`
`LOOP`
`UNTIL X>4 DO`
`  PRINT "X= ",X`
`  X=X+1`
`REPEAT`
	`X=  0`
	`X=  1`
	`X=  2`
	`X=  3`
	`X=  4`

`A=20`
`LOOP`
`  PRINT "A= ", A`
`  A=A-1`
`UNTIL A=15 REPEAT`
	`A=  20`
	`A=  19`
	`A=  18`
	`A=  17`
	`A=  16`

`Q=3`
`LOOP`
`  PRINT "Q= ",Q`
`WHILE Q DO`
`  Q=Q-1`
`REPEAT`
	`Q=  3`
	`Q=  2`
	`Q=  1`
	`Q=  0`

`EXECUTE "SELECT VOC FIRST 5"`
`MORE=1`
`LOOP`
`  READNEXT ID`
`  ELSE MORE=0`
`WHILE MORE DO`
`  PRINT ID`
`REPEAT`
	`5 record(s)`
	`selected to`
	`SELECT list #0.`
	`LOOP`
	`HASH.TEST`
	`QUIT.KEY`
	`P`
	`CLEAR.LOCKS`

Source Lines	Program Output

`EXECUTE "SELECT VOC FIRST 5"`
`LOOP`
`WHILE READNEXT ID DO`
`  PRINT ID`
`REPEAT`
	`5 record(s)`
	`selected to`
	`SELECT list #0.`
	`LOOP`
	`HASH.TEST`
	`QUIT.KEY`
	`P`
	`CLEAR.LOCKS`

Syntax

LOWER (*expression*)

Description

Use the LOWER function to return a value equal to *expression*, except that system delimiters which appear in *expression* are converted to the next lower-level delimiter: field marks are changed to value marks, value marks are changed to subvalue marks, and so on. If *expression* evaluates to the null value, null is returned.

The conversions are:

IM	CHAR(255)	to	FM	CHAR(254)
FM	CHAR(254)	to	VM	CHAR(253)
VM	CHAR(253)	to	SM	CHAR(252)
SM	CHAR(252)	to	TM	CHAR(251)
TM	CHAR(251)	to		CHAR(250)
	CHAR(250)	to		CHAR(249)
	CHAR(249)	to		CHAR(248)

PIOPEN Flavor

In PIOPEN flavor, the delimiters that can be lowered are CHAR(255) through CHAR(252). All other characters are left unchanged. You can obtain PIOPEN flavor for the LOWER function by:

- Compiling your program in a PIOPEN flavor account
- Specifying the $OPTIONS INFO.MARKS statement

Examples

In the following examples an item mark is shown by `I`, a field mark is shown by `F`, a value mark is shown by `V`, a subvalue mark is shown by `S`, and a text mark is shown by `T`. CHAR(250) is shown as `Z`.

The following example sets A to `DDFEEv123v77`:

```
A= LOWER('DD':IM'EE':FM:123:FM:777)
```

The next example sets B to `1F2S3V4T5`:

```
B= LOWER(1:IM:2:VM:3:FM:4:SM:5)
```

The next example sets C to 999z888:

```
C= LOWER(999:TM:888)
```

Syntax

LTS (*array1*, *array2*)

CALL –LTS (*return.array*, *array1*, *array2*)

CALL !LTS (*return.array*, *array1*, *array2*)

Description

Use the LTS function to test if elements of one dynamic array are less than elements of another dynamic array.

Each element of *array1* is compared with the corresponding element of *array2*. If the element from *array1* is less than the element from *array2*, a 1 is returned in the corresponding element of a new dynamic array. If the element from *array1* is greater than or equal to the element from *array2*, a 0 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, the undefined element is evaluated as an empty string, and the comparison continues.

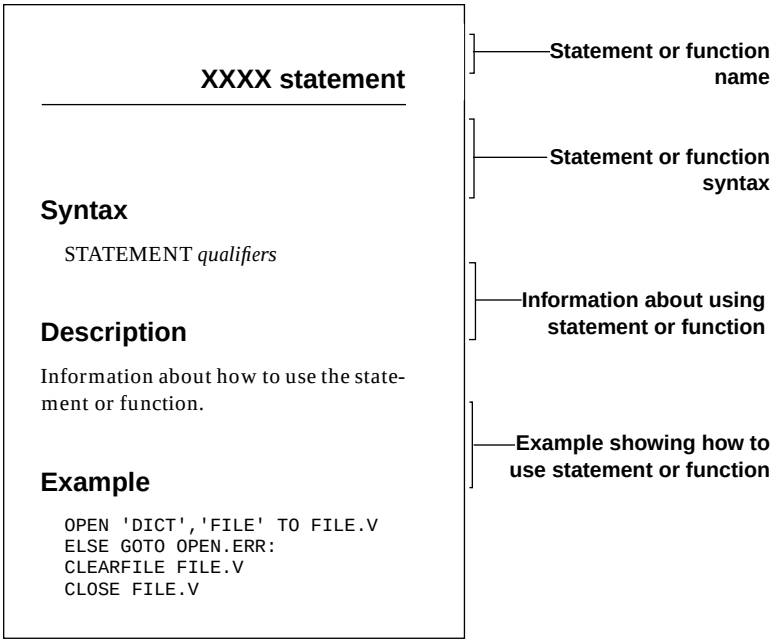
If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Statements and Functions M–X

This volume describes the UniVerse BASIC statements and functions whose names begin with the letters M through X.

Each statement or function is listed on a separate page. The sample shows a typical statement or function reference page.



MAT statement

Syntax

MAT *array* = *expression*

MAT *array1* = MAT *array2*

Description

Use the MAT statement to assign one value to all of the elements in the array or to assign all the values of one array to the values of another array.

Use the first syntax to assign the same value to all array elements. Use any valid expression. The value of *expression* becomes the value of each array element.

Use the second syntax to assign values from the elements of *array2* to the elements of *array1*. Both arrays must previously be named and dimensioned. The dimensioning of the two arrays can be different. The values of the elements of the new array are assigned in consecutive order, regardless of whether the dimensions of the arrays are the same or not. If *array2* has more elements than in *array1*, the extra elements are ignored. If *array2* has fewer elements, the extra elements of *array1* are not assigned.

Note: Do not use the MAT statement to assign individual elements of an array.

Examples

Source Lines	Program Output

`DIM ARRAY(5)`
`QTY=10`
`MAT ARRAY=QTY`
`FOR X=1 TO 5`
`  PRINT "ARRAY(":X:")=",ARRAY(X)`
`NEXT X`
	`ARRAY(1)=  10`
	`ARRAY(2)=  10`
	`ARRAY(3)=  10`
	`ARRAY(4)=  10`
	`ARRAY(5)=  10`

`DIM ONE(4,1)`
`MAT ONE=1`
`DIM TWO(2,2)`
`MAT TWO = MAT ONE`
`FOR Y=1 TO 4`
`  PRINT "ONE(":Y:",1)=",ONE(Y,1)`
`NEXT Y`
	`ONE(1,1)=  1`
	`ONE(2,1)=  1`
	`ONE(3,1)=  1`
	`ONE(4,1)=  1`

Source Lines	Program Output

`DIM ONE(4,1)`
`MAT ONE=1`
`DIM TWO(2,2)`
`MAT TWO = MAT ONE`
`FOR X=1 TO 2`
`  FOR Y=1 TO 2`
`  PRINT "TWO(":X:",":Y:")=",TWO(X,Y)`
`  NEXT Y`
`NEXT X`
	`TWO(1,1)=  1`
	`TWO(1,2)=  1`
	`TWO(2,1)=  1`
	`TWO(2,2)=  1`

The following example sets all elements in ARRAY to the empty string:

```
MAT ARRAY=''
```

MATBUILD statement

Syntax

MATBUILD *dynamic.array* FROM *array* [,*start* [,*end*]] [USING *delimiter*]

Description

Use the MATBUILD statement to build a dynamic array from a dimensioned array.

dynamic.array is created by concatenating the elements of *array* beginning with *start* and finishing with *end*. If *start* and *end* are not specified or are out of range, they default to 1 and the size of the array respectively.

array must be named and dimensioned in a DIMENSION or COMMON statement before it is used in this statement.

delimiter specifies characters to be inserted between fields of the dynamic array. If *delimiter* is not specified, it defaults to a field mark. To specify no delimiter, specify USING without *delimiter*.

If an element of *array* is the null value, the dynamic array will contain CHAR(128) for that element. If *start*, *end*, or *delimiter* is the null value, the MATBUILD statement fails, the program terminates, and a run-time error message appears.

Overflow Elements

PICK, IN2, and REALITY flavor dimensioned arrays contain overflow elements in the last element. INFORMATION and IDEAL flavor dimensioned arrays contain overflow elements in element 0.

In PICK, IN2, and REALITY flavor accounts, if *end* is not specified, *dynamic.array* contains the overflow elements of *array*. In IDEAL and INFORMATION flavor accounts, to get the overflow elements you must specify *end* as less than or equal to 0, or as greater than the size of *array*.

REALITY flavor accounts use only the first character of *delimiter*, and if USING is specified without a delimiter, *delimiter* defaults to a field mark rather than an empty string.

Syntax

string MATCH[ES] *pattern*

Description

Use the MATCH operator or its synonym MATCHES to compare a string expression with a pattern.

pattern is a general description of the format of *string*. It can consist of text or the special characters X, A, and N preceded by an integer used as a repeating factor. For example, *n*N is the pattern for strings of *n* numeric characters.

The following table lists the *pattern* codes and their definitions:

Pattern Matching Codes

Pattern	Definition
...	Any number of any characters (including none).
0X	Any number of any characters (including none).
*n*X	*n* number of any characters.
0A	Any number of alphabetic characters (including none).
nA	*n* number of alphabetic characters.
0N	Any number of numeric characters (including none).
*n*N	*n* number of numeric characters.
'*text*'	Exact text; any literal string (quotation marks required).
"*text*"	Exact text; any literal string (quotation marks required).

If *n* is longer than nine digits, it is used as text in a pattern rather than as a repeating factor for a special character. For example, the pattern "1234567890N" is treated as a literal string, not as a pattern of 1,234,567,890 numeric characters.

If the string being evaluated matches the pattern, the expression evaluates as true (1); otherwise, it evaluates as false (0). If either *string* or *pattern* is the null value, the match evaluates as false.

A tilde (~) placed immediately before *pattern* specifies a negative match. That is, it specifies a pattern or a part of a pattern that does not match the expression or a part of the expression. The match is true only if *string* and *pattern* are of equal

length and differ in at least one character. An example of a negative match pattern is:

```
"'A'~'X'5N
```

This pattern returns a value of true if the expression begins with the letter A, which is not followed by the letter X, and which is followed by any five numeric characters. Thus AB55555 matches the pattern, but AX55555, A55555, AX5555, and A5555 do not.

You can specify multiple patterns by separating them with value marks (ASCII CHAR(253)). The following expression is true if the address is either 16 alphabetic characters or 4 numeric characters followed by 12 alphabetic characters; otherwise, it is false:

```
ADDRESS MATCHES "16A": CHAR(253): "4N12A"
```

An empty string matches the following patterns: "0A", "0X", "0N", "...", "", ", or \\.

Syntax

MATCHFIELD (*string*, *pattern*, *field*)

Description

Use the MATCHFIELD function to check a string against a match pattern (see the MATCH operator for information about pattern matching).

field is an expression that evaluates to the portion of the match string to be returned.

If *string* matches *pattern*, the MATCHFIELD function returns the portion of *string* that matches the specified field in *pattern*. If *string* does not match *pattern*, or if *string* or *pattern* evaluates to the null value, the MATCHFIELD function returns an empty string. If field evaluates to the null value, the MATCHFIELD function fails, the program terminates, and a run-time error message appears.

pattern must contain specifiers to cover all characters contained in *string*. For example, the following statement returns an empty string because not all parts of *string* are specified in the pattern:

```
MATCHFIELD ("XYZ123AB", "3X3N", 1)
```

To achieve a positive pattern match on *string* above, the following statement might be used:

```
MATCHFIELD ("XYZ123AB", "3X3N0X", 1)
```

This statement returns a value of "XYZ".

Examples

Source Lines	Program Output
`Q=MATCHFIELD("AA123BBB9","2A0N3A0N",3)` `PRINT "Q= ",Q`	`Q=  BBB`

`ADDR='20 GREEN ST. NATICK, MA.,01234'`
`ZIP=MATCHFIELD(ADDR,"0N0X5N",3)`
`PRINT "ZIP= ",ZIP`
	`ZIP=  01234`

`INV='PART12345 BLUE AU'`
`COL=MATCHFIELD(INV,"10X4A3X",2)`
`PRINT "COL= ",COL`
	`COL=  BLUE`

In the following example the string does not match the pattern:

Source Lines	Program Output
`XYZ=MATCHFIELD('ABCDE1234',"2N3A4N",1)` `PRINT "XYZ= ",XYZ`	`XYZ=`

In the following example the entire string does not match the pattern:

Source Lines	Program Output
`ABC=MATCHFIELD('1234AB',"4N1A",2)` `PRINT "ABC= ",ABC`	`ABC=`

Syntax

MATPARSE *array* FROM *dynamic.array* [,*delimiter*]

MATPARSE *array* [,*start* [,*end*]] FROM *dynamic.array* [USING *delimiter*]
[SETTING *elements*]

Description

Use the MATPARSE statement to separate the fields of *dynamic.array* into consecutive elements of *array*.

array must be named and dimensioned in a DIMENSION or COMMON statement before it is used in this statement.

start specifies the starting position in *array*. If *start* is less than 1, it defaults to 1.

end specifies the ending position in *array*. If *end* is less than 1 or greater than the length of *array*, it defaults to the length of *array*.

delimiter is an expression evaluating to the characters used to delimit elements in *dynamic.array*. Use a comma or USING to separate delimiter from *dynamic.array*. *delimiter* can have no characters (an empty delimiter), one character, or more than one character with the following effects:

- An empty delimiter (a pair of quotation marks) parses *dynamic.array* so that each character becomes one element of *array* (see the second example). The default delimiter is a field mark. This is different from the empty delimiter. To use the default delimiter, omit the comma or USING following *dynamic.array*.
- A single character delimiter parses *dynamic.array* into fields delimited by that character by storing the substrings that are between successive delimiters as elements in the array. The delimiters are not stored in the array (see the first example).
- A multicharacter delimiter parses *dynamic.array* by storing as elements both the substrings that are between any two successive delimiters and the substrings consisting of one or more consecutive delimiters in the following way: *dynamic.array* is searched until any of the delimiter characters are found. All of the characters up to but not including the delimiter character are stored as an element of *array*. The delimiter character and any identical consecutive delimiter characters are stored as the next element.

The search then continues as at the start of *dynamic.array* (see the third example).

- If *delimiter* is a system delimiter and a single CHAR(128) is extracted from *dynamic.array*, the corresponding element in *array* is set to the null value.

The characters in a multicharacter delimiter expression can be different or the same. A delimiter expression of /: might be used to separate hours, minutes, seconds and month, day, year in the formats 12:32:16 and 1/23/85. A delimiter expression of two spaces " " might be used to separate tokens on a command line that contain multiple blanks between tokens.

The SETTING clause sets the variable *elements* to the number of elements in *array*. If *array* overflows, *elements* is set to 0. The value of *elements* is the same as the value returned by the INMAT function after a MATPARSE statement.

If all the elements of *array* are filled before MATPARSE reaches the end of *dynamic.array*, MATPARSE puts the unprocessed part of *dynamic.array* in the zero element of *array* for IDEAL, INFORMATION, or PIOPEN flavor accounts, or in the last element of *array* for PICK, IN2, or REALITY flavor accounts.

Use the INMAT function after a MATPARSE statement to determine the number of elements loaded into the array. If there are more delimited fields in *dynamic.array* than elements in *array*, INMAT returns 0; otherwise, it returns the number of elements loaded.

If *start* is greater than *end* or greater than the length of *array*, no action is taken, and INMAT returns 0.

If *start*, *end*, *dynamic.array*, or *delimiter* evaluates to the null value, the MATPARSE statement fails, the program terminates, and a run-time error message appears.

Examples

Source Lines	Program Output

`DIM X(4)`
`Y='1#22#3#44#5#66#7'`
`MATPARSE X FROM Y, '#'`
`FOR Z=0 TO 4`
`PRINT "X(":Z:")",X(Z)`
`NEXT Z`
`PRINT`
	`X(0)  5#66#7`
	`X(1)  1`
	`X(2)  22`
	`X(3)  3`
	`X(4)  44`

Source Lines	Program Output

`DIM Q(6)`
`MATPARSE Q FROM 'ABCDEF', ''`
`FOR P=1 TO 6`
`PRINT "Q(":P:")",Q(P)`
`NEXT P`
`PRINT`
	`Q(1)  A`
	`Q(2)  B`
	`Q(3)  C`
	`Q(4)  D`
	`Q(5)  E`
	`Q(6)  F`

`DIM A(8,2)`
`MATPARSE A FROM 'ABCDEFGDDDHIJCK', 'CD'`
`FOR I = 1 TO 8`
`FOR J = 1 TO 2`
`PRINT "A(":I:",":J:")=",A(I,J)," ":`
`NEXT J`
`PRINT`
`NEXT I`
`END`
	`A(1,1)= AB  A(1,2)= C`
	`A(2,1)=    A(2,2)= D`
	`A(3,1)= EFG  A(3,2)= DDD`
	`A(4,1)= HIJ  A(4,2)= C`
	`A(5,1)= K  A(5,2)=`
	`A(6,1)=    A(6,2)=`
	`A(7,1)=    A(7,2)=`
	`A(8,1)=    A(8,2)=`

MATREAD statements

Syntax

MATREAD *array* FROM [*file.variable,*] *record.ID* [ON ERROR *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

{ MATREADL | MATREADU } *array* FROM [*file.variable,*] *record.ID*
[ON ERROR *statements*] [LOCKED *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the MATREAD statement to assign the contents of the fields of a record from a UniVerse file to consecutive elements of *array*. The first field of the record becomes the first element of *array*, the second field of the record becomes the second element of *array*, and so on. The array must be named and dimensioned in a DIMENSION or COMMON statement before it is used in this statement.

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information about default files, see the OPEN statement). If the file is neither accessible nor open, the program terminates and a run-time message appears.

If *record.ID* exists, *array* is set to the contents of the record, and the THEN statements are executed; any ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next sequential statement. If *record.ID* does not exist, the elements of *array* are not changed, and the ELSE statements are executed; any THEN statements are ignored.

If either *file.variable* or *record.ID* evaluates to the null value, the MATREAD statement fails, the program terminates, and a run-time error message appears. If any field in the record is the null value, null becomes an element in *array*. If a value or a subvalue in a multivalued field is the null value, it is read into the field as the stored representation of null (CHAR(128)).

If the file is an SQL table, the effective user of the program must have SQL SELECT privilege to read records in the file. For information about the effective user of a program, see the AUTHORIZATION statement in Volume 1.

A MATREAD statement does not set an update record lock on the specified record. That is, the record remains available for update to other users. To prevent other users from updating the record until it is released, use a MATREADL or MATREADU statement.

If the number of elements in *array* is greater than the number of fields in the record, the extra elements in *array* are assigned empty string values. If the number of fields in the record is greater than the number of elements in the array, the extra values are stored in the zero element of *array* for IDEAL or INFORMATION flavor accounts, or in the last element of *array* for PICK, IN2, or REALITY flavor accounts. The zero element of an array can be accessed with a 0 subscript as follows:

```
MATRIX (0)
```

or

```
MATRIX (0, 0)
```

Use the INMAT function after a MATREAD statement to determine the number of elements of the array that were actually used. If the number of fields in the record is greater than the number of elements in the array, the value of the INMAT function is set to 0.

MATREADL and MATREADU Statements

Use the MATREADL syntax to acquire a shared record lock and then perform a MATREAD. This lets other programs read the record with no lock or a shared record lock.

Use the MATREADU syntax to acquire an update record lock and then perform a MATREAD. The update record lock prevents other users from updating the record until the user who set it releases it.

An update record lock can be acquired when no shared record lock exists, or promoted from a shared record lock owned by you if no other shared record locks exist.

The ON ERROR Clause

The ON ERROR clause is optional in MATREAD statements. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the MATREAD statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.

- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED Clause

The LOCKED clause is optional, but recommended. Its syntax is the same as that of the ELSE clause.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the MATREAD statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

In this statement...	This requested lock...	Conflicts with these locks...
MATREADL MATREADVL	Shared record	Exclusive file Update record
MATREADU MATREADVU	Update record	Exclusive file Intent file Shared file Update record Shared record

If a MATREAD statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

Releasing Locks

A shared record lock can be released with a CLOSE, RELEASE, or STOP statement. An update record lock can be released with a CLOSE, DELETE, MATWRITE, RELEASE, STOP, WRITE, or WRITEV statement.

Locks acquired or promoted within a transaction are not released when the previous statements are processed.

Example

```
DIM ARRAY(10)
OPEN 'SUN.MEMBER' TO SUN.MEMBER ELSE STOP
MATREAD ARRAY FROM SUN.MEMBER, 6100 ELSE STOP
*
FOR X=1 TO 10
PRINT "ARRAY(":X:")",ARRAY(X)
NEXT X
*
PRINT
*
DIM TEST(4)
OPEN '','SUN.SPORT' ELSE STOP 'CANNOT OPEN SUN.SPORT'
MATREAD TEST FROM 851000 ELSE STOP
*
FOR X=0 TO 4
PRINT "TEST(":X:")",TEST(X)
NEXT X
```

This is the program output:

```
ARRAY(1) MASTERS
ARRAY(2) BOB
ARRAY(3) 55 WESTWOOD ROAD
ARRAY(4) URBANA
ARRAY(5) IL
ARRAY(6) 45699
ARRAY(7) 1980
ARRAY(8) SAILING
ARRAY(9)
ARRAY(10)     II
TEST(0) 6258
TEST(1) 6100
TEST(2) HARTWELL
TEST(3) SURFING
TEST(4) 4
```

MATREADL statement

Use the MATREADL statement to set a shared record lock and perform the MATREAD statement. For details, see the MATREAD statement.

MATREADU statement

Use the MATREADU statement to set an update record lock and perform the MATREAD statement. For details, see the MATREAD statement.

MATWRITE statements

Syntax

MATWRITE[U] *array* ON | TO [*file.variable,*] *record.ID*
[ON ERROR *statements*] [LOCKED *statements*]
[THEN *statements*] [ELSE *statements*]

Description

Use the MATWRITE statement to write data from the elements of a dimensioned array to a record in a UniVerse file. The elements of *array* replace any data stored in the record. MATWRITE strips any trailing empty fields from the record.

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement). If the file is neither accessible nor open, the program terminates and a run-time message appears, unless ELSE statements are specified.

If the file is an SQL table, the effective user of the program must have SQL INSERT and UPDATE privileges to MATWRITE records in the file. See the AUTHORIZATION statement in Volume 1 for information about the effective user of a program.

If the OPENCHK configurable parameter is set to TRUE, or if the file is opened with the OPENCHECK statement, all SQL integrity constraints are checked for every MATWRITE to an SQL table. If an integrity check fails, the MATWRITE statement uses the ELSE clause. Use the ICHECK function to determine what specific integrity constraint caused the failure.

The system searches the file for the record specified by *record.ID*. If the record is not found, MATWRITE creates a new record.

The ON ERROR Clause

The ON ERROR clause is optional in the MATWRITE statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the MATWRITE is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.

- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED Clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the MATWRITE statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file
- Intent file
- Shared file
- Update record
- Shared record

If the MATWRITE statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

When updating a file, MATWRITE releases the update record lock set with a MATREADU statement. To maintain the update record lock set with the MATREADU statement, use MATWRITEU instead of MATWRITE.

The new values are written to the record, and the THEN clauses are executed. If no THEN statements are specified, execution continues with the statement following the MATWRITE statement.

If either *file.variable* or *record.ID* evaluates to the null value, the MATWRITE statement fails, the program terminates, and a run-time error message appears. Null elements of *array* are written to *record.ID* as the stored representation of the null value, CHAR(128).

IDEAL and INFORMATION Flavors

In IDEAL and INFORMATION flavor accounts, if the zero element of the array has been assigned a value by a MATREAD or MATREADU statement, the zero element value is written to the record as the *n*+1 field, where *n* is the number of elements dimensioned in the array. If the zero element is assigned an empty string, only the assigned elements of the array are written to the record; trailing empty fields are ignored. The new record is written to the file (replacing any existing record) without regard for the size of the array.

It is generally good practice to use the MATWRITE statement with arrays that have been loaded with either a MATREAD or a MATREADU statement.

After executing a MATWRITE statement, you can use the STATUS function to determine the result of the operation as follows (see the STATUS function for more information):

0 The record was locked before the MATWRITE operation.

–2 The record was unlocked before the MATWRITE operation.

–3 The record failed an SQL integrity check.

The MATWRITEU Statement

Use the MATWRITEU statement to update a record without releasing the update record lock set by a previous MATREADU statement (see the MATREAD statement). To release the update record lock set by a MATREADU statement and maintained by a MATWRITEU statement, you must use a RELEASE or MATWRITE statement. If you do not explicitly release the lock, the record remains locked until the program executes the STOP statement. When more than one program or user could modify the same record, use a MATREADU statement to lock the record before doing the MATWRITE or MATWRITEU.

Example

```
DIM ARRAY(5)
OPEN 'EX.BASIC' TO EX.BASIC ELSE STOP 'CANNOT OPEN'
MATREADU ARRAY FROM EX.BASIC, 'ABS' ELSE STOP
ARRAY(1)='Y = 100'
MATWRITE ARRAY TO EX.BASIC, 'ABS'
PRINT 'STATUS()= ',STATUS()
```

This is the program output:

```
STATUS()=     0
```

MATWRITEU statement

Use the MATWRITEU statement to maintain an update record lock and perform the MATWRITE statement. For details, see the MATWRITE statement.

MAXIMUM function

Syntax

MAXIMUM (*dynamic.array*)

CALL !MAXIMUM (*result*, *dynamic.array*)

Description

Use the MAXIMUM function to return the element with the highest numeric value in *dynamic.array.* Nonnumeric values, except the null value, are treated as 0. If *dynamic.array* evaluates to the null value, null is returned. Any element that is the null value is ignored, unless all elements of *dynamic.array* are null, in which case null is returned.

result is the variable that contains the largest element found in *dynamic.array.*

dynamic.array is the array to be tested.

Examples

```
A=1:@VM:"ZERO":@SM:20:@FM:-25
PRINT "MAX(A)=",MAXIMUM(A)
```

This is the program output:

```
MAX(A)=20
```

In the following example, the !MAXIMUM subroutine is used to obtain the maximum value contained in array A. The nonnumeric value, Z, is treated as 0.

```
A=1:@VM:25:@VM:'Z':@VM:7
CALL !MAXIMUM (RESULT,A)
PRINT RESULT
```

This is the program output:

```
0
```

Syntax

MINIMUM (*dynamic.array*)

CALL !MINIMUM (*result*, *dynamic.array*)

Description

Use the MINIMUM function to return the element with the lowest numeric value in *dynamic.array.* Nonnumeric values, except the null value, are treated as 0. If *dynamic.array* evaluates to the null value, null is returned. Any element that is the null value is ignored, unless all elements of *dynamic.array* are null, in which case null is returned.

result is the variable that contains the smallest element found in *dyamic.array.*

dynamic.array is the array to be tested.

Examples

```
A=1:@VM:"ZERO":@SM:20:@FM:-25
PRINT "MIN(A)=",MINIMUM(A)
```

This is the program output:

```
MIN(A)=-25
```

In the following example, the !MINIMUM subroutine is used to obtain the minimum value contained in array A. The nonnumeric value, Q, is treated as 0.

```
A=2:@VM:19:@VM:6:@VM:'Q'
CALL !MINIMUM (RESULT,A)
PRINT RESULT
```

This is the program output:

```
0
```

MOD function

Syntax

MOD (*dividend*, *divisor*)

Description

Use the MOD function to calculate the value of the remainder after integer division is performed on the dividend expression by the divisor expression.

The MOD function calculates the remainder using the following formula:

MOD (X, Y) = X – (INT (X / Y) * Y)

dividend and *divisor* can evaluate to any numeric value, except that *divisor* cannot be 0. If *divisor* is 0, a division by 0 warning message is printed, and 0 is returned. If either *dividend* or *divisor* evaluates to the null value, null is returned.

The MOD function works like the REM function.

Example

```
X=85; Y=3
PRINT 'MOD (X,Y)= ',MOD (X,Y)
```

This is the program output:

```
MOD (X,Y)=     1
```

Syntax

MODS (*array1*, *array2*)

CALL –MODS (*return.array*, *array1*, *array2*)

CALL !MODS (*return.array*, *array1*, *array2*)

Description

Use the MODS function to create a dynamic array of the remainder after the integer division of corresponding elements of two dynamic arrays.

The MODS function calculates each element according to the following formula:

XY.element = X – (INT (X / Y) * Y)

X is an element of *array1* and Y is the corresponding element of *array2*. The resulting element is returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, 0 is returned. If an element of *array2* is 0, 0 is returned. If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A=3:@VM:7
B=2:@SM:7:@VM:4
PRINT MODS(A,B)
```

This is the program output:

```
1s0v3
```

MULS function

Syntax

MULS (*array1*, *array2*)

CALL –MULS (*return.array*, *array1*, *array2*)

CALL !MULS (*return.array*, *array1*, *array2*)

Description

Use the MULS function to create a dynamic array of the element-by-element multiplication of two dynamic arrays.

Each element of *array1* is multiplied by the corresponding element of *array2* with the result being returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, 0 is returned. If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A=1:@VM:2:@VM:3:@SM:4
B=4:@VM:5:@VM:6:@VM:9
PRINT MULS(A,B)
```

This is the program output:

```
4v10v18s0v0
```

Syntax

NAP [*milliseconds*]

Description

Use the NAP statement to suspend the execution of a BASIC program, pausing for a specified number of milliseconds.

milliseconds is an expression evaluating to the number of milliseconds for the pause. If *milliseconds* is not specified, a value of 1 is used. If *milliseconds* evaluates to the null value, the NAP statement is ignored.

NEG function

Syntax

NEG (*number*)

Description

Use the NEG function to return the arithmetic inverse of the value of the argument.

number is an expression evaluating to a number.

Example

In the following example, A is assigned the value of 10, and B is assigned the value of NEG(A), which evaluates to –10:

```
A = 10
B = NEG(A)
```

NEGS function

Syntax

NEGS (*dynamic.array*)

CALL –NEGS (*return.array*, *dynamic.array*)

Description

Use the NEGS function to return the negative values of all the elements in a dynamic array. If the value of an element is negative, the returned value is positive. If *dynamic.array* evaluates to the null value, null is returned. If any element is null, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

NES function

Syntax

NES (*array1*, *array2*)

CALL –NES (*return.array*, *array1*, *array2*)

CALL !NES (*return.array*, *array1*, *array2*)

Description

Use the NES function to test if elements of one dynamic array are equal to the elements of another dynamic array.

Each element of *array1* is compared with the corresponding element of *array2*. If the two elements are equal, a 0 is returned in the corresponding element of a new dynamic array. If the two elements are not equal, a 1 is returned. If an element of one dynamic array has no corresponding element in the other dynamic array, a 1 is returned. If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Syntax

NEXT [*variable*]

Description

Use the NEXT statement to end a FOR...NEXT loop, causing the program to branch back to the FOR statement and execute the statements that follow it.

Each FOR statement must have exactly one corresponding NEXT statement.

variable is the name of the variable given as the index counter in the FOR statement. If the variable is not named, the most recently named index counter variable is assumed.

Example

```
FOR I=1 TO 10
   PRINT I:" ":
NEXT I
PRINT
```

This is the program output:

```
1 2 3 4 5 6 7 8 9 10
```

NOBUF statement

Syntax

NOBUF *file.variable* {THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the NOBUF statement to turn off buffering for a file previously opened for sequential processing. Normally UniVerse uses buffering for sequential input and output operations. The NOBUF statement turns off this buffering and causes all writes to the file to be performed immediately. It eliminates the need for FLUSH operations but also eliminates the benefits of buffering. The NOBUF statement must be executed after a successful OPENSEQ or CREATE statement and before any input or output operation is performed on the record.

If the NOBUF operation is successful, the THEN statements are executed; the ELSE statements are ignored. If THEN statements are not present, program execution continues with the next statement.

If the specified file cannot be accessed or does not exist, the ELSE statements are executed; the THEN statements are ignored. If *file.variable* evaluates to the null value, the NOBUF statement fails, the program terminates, and a run-time error message appears.

Example

In the following example, if RECORD1 in FILE.E can be opened, buffering is turned off:

```
OPENSEQ 'FILE.E', 'RECORD1' TO DATA THEN NOBUF DATA
ELSE ABORT
```

Syntax

NOT (*expression*)

Description

Use the NOT function to return the logical complement of the value of *expression*. If the value of *expression* is true, the NOT function returns a value of false (0). If the value of *expression* is false, the NOT function returns a value of true (1).

A numeric expression that evaluates to 0 is a logical value of false. A numeric expression that evaluates to anything else, other than the null value, is a logical true.

An empty string is logically false. All other string expressions, including strings that include an empty string, spaces, or the number 0 and spaces, are logically true.

If *expression* evaluates to the null value, null is returned.

Example

```
X=5; Y=5
PRINT NOT(X-Y)
PRINT NOT(X+Y)
```

This is the program output:

```
1
0
```

NOTS function

Syntax

NOTS (*dynamic.array*)

CALL –NOTS (*return.array, dynamic.array*)

CALL !NOTS (*return.array, dynamic.array*)

Description

Use the NOTS function to return a dynamic array of the logical complements of each element of *dynamic.array*. If the value of the element is true, the NOTS function returns a value of false (0) in the corresponding element of the returned array. If the value of the element is false, the NOTS function returns a value of true (1) in the corresponding element of the returned array.

A numeric expression that evaluates to 0 has a logical value of false. A numeric expression that evaluates to anything else, other than the null value, is a logical true.

An empty string is logically false. All other string expressions, including strings which consist of an empty string, spaces, or the number 0 and spaces, are logically true.

If any element in *dynamic.array* is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
X=5; Y=5
PRINT NOTS(X-Y:@VM:X+Y)
```

This is the program output:

```
1v0
```

Syntax

NULL

Description

Use the NULL statement when a statement is required but no operation is to be performed. For example, you can use it with the ELSE clause if you do not want any operation performed when the ELSE clause is executed.

Note: This statement has nothing to do with the null value.

Example

```
OPEN '','SUN.MEMBER' TO FILE ELSE STOP
FOR ID=5000 TO 6000
   READ MEMBER FROM FILE, ID THEN PRINT ID ELSE NULL
NEXT ID
```

NUM function

Syntax

NUM (*expression*)

Description

Use the NUM function to determine whether *expression* is a numeric or nonnumeric string. If *expression* is a number, a numeric string, or an empty string, it evaluates to true and a value of 1 is returned. If *expression* is a nonnumeric string, it evaluates to false and a value of 0 is returned.

The NUM of a numeric string with a decimal point (.) evaluates to true; the NUM of a numeric string with a comma (,) or dollar sign ($) evaluates to false.

If *expression* evaluates to the null value, null is returned.

Example

```
X=NUM(2400)
Y=NUM("Section 4")
PRINT "X= ",X,"Y= ",Y
```

This is the program output:

```
X=    1     Y=    0
```

Syntax

NUMS (*dynamic.array*)

CALL –NUMS (*return.array, dynamic.array*)

CALL !NUMS (*return.array, dynamic.array*)

Description

Use the NUMS function to determine whether the elements of a dynamic array are numeric or nonnumeric strings. If an element is numeric, a numeric string, or an empty string, it evaluates to true, and a value of 1 is returned to the corresponding element in a new dynamic array. If the element is a nonnumeric string, it evaluates to false, and a value of 0 is returned.

The NUMS of a numeric element with a decimal point (.) evaluates to true; the NUMS of a numeric element with a comma (,) or dollar sign ($) evaluates to false.

If *dynamic.array* evaluates to the null value, null is returned. If an element of *dynamic.array* is null, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

OCONV function

Syntax

OCONV (*expression*, *conversion*)

Description

Use the OCONV function to convert *expression* to a specified format for external output. The result is always a string expression.

expression is converted to the external output format specified by *conversion*.

conversion must evaluate to one or more conversion codes separated by value marks (ASCII 253).

If multiple codes are used, they are applied from left to right as follows: the left-most conversion code is applied to *expression*, the next conversion code to the right is then applied to the result of the first conversion, and so on.

If *expression* evaluates to the null value, null is returned. If *conversion* evaluates to the null value, the OCONV function fails, the program terminates, and a run-time error message appears.

The OCONV function also allows PICK flavor exit codes.

The STATUS function reflects the result of the conversion as follows:

0 The conversion is successful.

1 An invalid string is passed to the OCONV function; the original string is returned as the value of the conversion. If the invalid string is the null value, null is returned.

2 The conversion code is invalid.

See the STATUS function for more information.

For information about converting expressions to an internal format, see the ICONV function.

Examples

The following examples show date conversions:

Source Line	Converted Value
DATE=OCONV('9166',"D2")	3 Feb 93
DATE=OCONV(9166,'D/E')	3/2/1993
DATE=OCONV(9166,'DI')[1]	3/2/1993
DATE=OCONV('9166',"D2-")	2-3-93
DATE=OCONV(0,'D')	31 Dec 1967

1. For IN2, PICK, and REALITY flavor accounts.

The following examples show time conversions:

Source Line	Converted Value
TIME=OCONV(10000,"MT")	02:46
TIME=OCONV("10000","MTHS")	02:46:40am
TIME=OCONV(10000,"MTH")	02:46am
TIME=OCONV(10000,"MT.")	02.46
TIME=OCONV(10000,"MTS")	02:46:40

The following examples show hex, octal, and binary conversions:

Source Line	Converted Value
HEX=OCONV(1024,"MX")	400
HEX=OCONV('CDE',"MX0C")	434445
OCT=OCONV(1024,"MO")	2000
OCT=OCONV('CDE',"MO0C")	103104105

Source Line	Converted Value
BIN=OCONV(1024,"MB")	10000000000
BIN=OCONV('CDE',"MB0C")	010000110100010001000101

The following examples show masked decimal conversions:

Source Line	Converted Value
X=OCONV(987654,"MD2")	9876.54
X=OCONV(987654,"MD0")	987654
X=OCONV(987654,"MD2,$")	$9,876.54
X=OCONV(987654,"MD24$")	$98.77
X=OCONV(987654,"MD2-Z")	9876.54
X=OCONV(987654,"MD2,D")	9,876.54
X=OCONV(987654,"MD3,$CPZ")	$987.654
X=OCONV(987654,"MD2,ZP12#")	####9,876.54

Syntax

OCONVS (*dynamic.array, conversion*)

CALL –OCONVS (*return.array, dynamic.array, conversion*)

CALL !OCONVS (*return.array, dynamic.array, conversion*)

Description

Use the OCONVS function to convert the elements of *dynamic.array* to a specified format for external output.

The elements are converted to the external output format specified by *conversion* and returned in a dynamic array. *conversion* must evaluate to one or more conversion codes separated by value marks (ASCII 253).

If multiple codes are used, they are applied from left to right as follows: the leftmost conversion code is applied to the element, the next conversion code to the right is then applied to the result of the first conversion, and so on.

If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is null, null is returned for that element. If *conversion* evaluates to the null value, the OCONVS function fails, the program terminates, and a run-time error message appears.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

The STATUS function reflects the result of the conversion as follows:

0 The conversion is successful.

1 An invalid element is passed to the OCONVS function; the original element is returned. If the invalid element is the null value, null is returned for that element.

2 The conversion code is invalid.

See the STATUS function for more information.

For information about converting elements in a dynamic array to an internal format, see the ICONVS function.

ON statement

Syntax

ON *expression* GOSUB *statement.label* [:] [,*statement.label* [:] ...]

ON *expression* GO[TO] *statement.label* [:] [,*statement.label* [:] ...]

Description

Use the ON statement to transfer program control to one of the internal subroutines named in the GOSUB clause or to one of the statements named in the GOTO clause.

Using the GOSUB Clause

Use ON GOSUB to transfer program control to one of the internal subroutines named in the GOSUB clause. The value of *expression* in the ON clause determines which of the subroutines named in the GOSUB clause is to be executed.

During execution, *expression* is evaluated and rounded to an integer. If the value of *expression* is 1 or less than 1, the first subroutine named in the GOSUB clause is executed; if the value of *expression* is 2, the second subroutine is executed; and so on. If the value of *expression* is greater than the number of subroutines named in the GOSUB clause, the last subroutine is executed. If *expression* evaluates to the null value, the ON statement fails, the program terminates, and a run-time error message appears.

statement.label can be any valid label defined in the program. If a nonexistent statement label is given, an error message is issued when the program is compiled. You must use commas to separate statement labels. You can use a colon with the statement labels to distinguish them from variable names.

A RETURN statement in the subroutine returns program flow to the statement following the ON GOSUB statement.

The ON GOSUB statement can be written on more than one line. A comma is required at the end of each line of the ON GOSUB statement except the last.

Using ON GOSUB in a PICK Flavor Account

If the value of *expression* is less than 1, the next statement is executed; if the value of *expression* is greater than the number of subroutines named in the GOSUB clause, execution continues with the next statement rather than the last subroutine.

To get this characteristic in other flavors, use the ONGO.RANGE option of the $OPTIONS statement.

Using the GOTO Clause

Use ON GOTO to transfer program control to one of the statements named in the GOTO clause. The value of *expression* in the ON clause determines which of the statements named in the GOTO clause is to be executed. During execution, *expression* is evaluated and rounded to an integer.

If the value of *expression* is 1 or less than 1, control is passed to the first statement label named in the GOTO clause; if the value of *expression* is 2, control is passed to the second statement label; and so on. If the value of *expression* is greater than the number of statements named in the GOTO clause, control is passed to the last statement label. If *expression* evaluates to the null value, the ON statement fails, the program terminates, and a run-time error message appears.

statement.label can be any valid label defined in the program. If a nonexistent statement label is given, an error message is issued when the program is compiled. You must use commas to separate statement labels. You can use a colon with the statement labels to distinguish them from variable names.

Using ON GOTO in a PICK Flavor Account

If the value of *expression* is less than 1, control is passed to the next statement; if the value of *expression* is greater than the number of the statements named in the GOTO clause, execution continues with the next statement rather than the last statement label. To get this characteristic with other flavors, use the ONGO.RANGE option of the $OPTIONS statement.

Examples

Source Lines	Program Output

`FOR X=1 TO 4`
`ON X GOSUB 10,20,30,40`
`PRINT 'RETURNED FROM SUBROUTINE'`
`NEXT X`
`STOP`
`10 PRINT 'AT LABEL 10'`
`RETURN`
`20 PRINT 'AT LABEL 20'`
`RETURN`
`30 PRINT 'AT LABEL 30'`
`RETURN`
`40 PRINT 'AT LABEL 40'`
`RETURN`
	`AT LABEL 10`
	`RETURNED FROM SUBROUTINE`
	`AT LABEL 20`
	`RETURNED FROM SUBROUTINE`
	`AT LABEL 30`
	`RETURNED FROM SUBROUTINE`
	`AT LABEL 40`
	`RETURNED FROM SUBROUTINE`

`VAR=1234`
`Y=1`
`10*`
`X=VAR[Y,1]`
`IF X='' THEN STOP`
`ON X GOTO 20,30,40`
`20*`
`PRINT 'AT LABEL 20'`
`Y=Y+1`
`GOTO 10`
`30*`
`PRINT 'AT LABEL 30'`
`Y=Y+1`
`GOTO 10`
`40*`
`PRINT 'AT LABEL 40'`
`Y=Y+1`
`GOTO 10`
	`AT LABEL 20`
	`AT LABEL 30`
	`AT LABEL 40`
	`AT LABEL 40`

Syntax

OPEN [*dict*,] *filename* [TO *file.variable*][ON ERROR *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the OPEN statement to open a UniVerse file for use by BASIC programs. All file references in a BASIC program must be preceded by either an OPEN statement or an OPENCHECK statement for that file. You can open several UniVerse files at the same point in a program, but you must use a separate OPEN statement for each file.

dict is an expression that evaluates to a string specifying whether to open the file dictionary or the data file. Use the string DICT to open the file dictionary, or use PDICT to open an associated Pick-style dictionary. Any other string opens the data file. By convention an empty string or the string DATA is used when you are opening the data file. If the *dict* expression is omitted, the data file is opened. If *dict* is the null value, the data file is opened.

filename is an expression that evaluates to the name of the file to be opened. If the file exists, the file is opened, and the THEN statements are executed; the ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next statement. If the file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored. If *filename* evaluates to the null value, the OPEN statement fails, the program terminates, and a run-time error message appears.

Use the TO clause to assign the opened file to *file.variable*. All statements that read, write to, delete, or clear that file must refer to it by the name of the file variable. If you do not assign the file to a file variable, an internal default file variable is used. File references that do not specify a file variable access the default file variable, which contains the most recently opened file. The file opened to the current default file variable is assigned to the system variable @STDFIL.

Default file variables are not local to the program from which they are executed. When a subroutine is called, the current default file variable is shared with the calling program.

When opening an SQL table, the OPEN statement enforces SQL security. The permissions granted to the program's effective user ID are loaded when the file is

opened. If no permissions have been granted, the OPEN statement fails, and the ELSE statements are executed.

All writes to an SQL table opened with the OPEN statement are subject to SQL integrity checking unless the OPENCHK configurable parameter has been set to FALSE. Use the OPENCHECK statement instead of the OPEN statement to enable automatic integrity checking for all writes to a file, regardless of whether the OPENCHK configurable parameter is true or false.

Use the INMAT function after an OPEN statement to determine the modulo of the file.

The ON ERROR Clause

The ON ERROR clause is optional in the OPEN statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the OPEN statement is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The STATUS Function

The file type is returned if the file is opened successfully. If the file is not opened successfully, the following values may return:

Value	Description
–1	Filename not found in the VOC file.
–2[1]	Null filename or file. This error may also occur when you cannot open a file across UV/Net.
–3	Operating system access error that occurs when you do not have permission to access a UniVerse file in a directory. For example, this may occur when trying to access a type 1 or type 30 file.
–4[1]	Access error when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
–5	Read error detected by the operating system.
–6	Unable to lock file header.
–7	Invalid file revision or wrong byte-ordering for the platform.
–8[1]	Invalid part file information.
–9[1]	Invalid type 30 file information in a distributed file.
–10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked “inconsistent.”
–11	The file is a view, therefore it cannot be opened by a BASIC program.
–12	No SQL privileges to open the table.
–13[1]	Index problem.
–14	Cannot open NFS file.

1. A generic error that can occur for various reasons.

Examples

```
OPEN "SUN.MEMBER" TO DATA ELSE STOP "CAN'T OPEN SUN.MEMBER"
OPEN "FOOBAR" TO FOO ELSE STOP "CAN'T OPEN FOOBAR"
PRINT "ALL FILES OPEN OK"
```

This is the program output:

```
CAN'T OPEN FOOBAR
```

The following example opens the same file as in the previous example. The OPEN statement includes an empty string for the *dict* argument.

```
OPEN "","SUN.MEMBER" TO DATA ELSE STOP "CAN'T OPEN SUN.MEMBER"
OPEN "","FOO.BAR" TO FOO ELSE STOP "CAN'T OPEN FOOBAR"
PRINT "ALL FILES OPEN OK"
```

Syntax

OPENCHECK [*dict*,] *filename* [TO *file.variable*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the OPENCHECK statement to open an SQL table for use by BASIC programs, enforcing SQL integrity checking. All file references in a BASIC program must be preceded by either an OPENCHECK statement or an OPEN statement for that file.

The OPENCHECK statement works like the OPEN statement, except that SQL integrity checking is enabled if the file is an SQL table. All field integrity checks for an SQL table are stored in the security and integrity constraints area (SICA). The OPENCHECK statement loads the compiled form of these integrity checks into memory, associating them with the file variable. All writes to the file are subject to SQL integrity checking.

The STATUS Function

The file type is returned if the file is opened successfully. If the file is not opened successfully, the following values may return:

Value	Description
–1	Filename not found in the VOC file.
–2[1]	Null filename or file. This error may also occur when you cannot open a file across UV/Net.
–3	Operating system access error that occurs when you do not have permission to access a UniVerse file in a directory. For example, this may occur when trying to access a type 1 or type 30 file.
–4[1]	Access error when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
–5	Read error detected by the operating system.
–6	Unable to lock file header.
–7	Invalid file revision or wrong byte-ordering for the platform.
–8[1]	Invalid part file information.
–9[1]	Invalid type 30 file information in a distributed file.

Value	Description
–10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked “inconsistent.”
–11	The file is a view, therefore it cannot be opened by a BASIC program.
–12	No SQL privileges to open the table.
–13[1]	Index problem.
–14	Cannot open NFS file.

1. A generic error that can occur for various reasons.

Syntax

OPENDEV *device* TO *file.variable* [LOCKED *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the OPENDEV statement to open a device for sequential processing. OPENDEV also sets a record lock on the opened device or file. See the READSEQ and WRITESEQ statements for more details on sequential processing.

device is an expression that evaluates to the record ID of a device definition record in the &DEVICE& file. If *device* evaluates to the null value, the OPENDEV statement fails, the program terminates, and a run-time error message appears. For more information, see "Devices on Windows NT."

The TO clause assigns a *file.variable* to the device being opened. All statements used to read to or write from that device must refer to it by the assigned *file.variable*.

If the device exists and is not locked, the device is opened and any THEN statements are executed; the ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next statement.

If the device is locked, the LOCKED statements are executed; THEN statements and ELSE statements are ignored.

If the device does not exist or cannot be opened, the ELSE statements are executed; any THEN statements are ignored. The device must have the proper access permissions for you to open it.

Devices on Windows NT

On Windows NT systems, you may need to change to block size defined for a device in the &DEVICE& file before you can use the OPENDEV statement to reference the device. On some devices there are limits to the type of sequential processing that is available once you open the device. The following table summarizes the limits:

Device Type	Block Size	Processing Available
4 mm DAT drive	No change needed.	No limits.
8 mm DAT drive	No change needed.	No limits.

Device Type	Block Size	Processing Available
1/4-inch cartridge drive, 60 MB or 150 MB	Specify the block size as 512 bytes or a multiple of 512 bytes.	Use READBLK and WRITEBLK to read or write data in blocks of 512 bytes. Use SEEK only to move the file pointer to the beginning or the end of the file. You can use WEOF to write an end-of-file (EOF) mark only at the beginning of the data or after a write.
1/4-inch 525 cartridge drive	No change needed.	No limits.
Diskette drive	Specify the block size as 512 bytes or a multiple of 512 bytes.	Use SEEK only to move the file pointer to the beginning of the file. Do not use WEOF.

The LOCKED Clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the OPENDEV statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file
- Intent file
- Shared file
- Update record
- Shared record

If the OPENDEV statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

Example

The following example opens TTY30 for sequential input and output operations:

```
OPENDEV 'TTY30' TO TERM THEN PRINT 'TTY30 OPENED'
ELSE ABORT
```

This is the program output:

```
TTY30 OPENED
```

Syntax

OPENPATH *pathname* [TO *file.variable*] [ON ERROR *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

The OPENPATH statement is similar to the OPEN statement, except that the pathname of the file is specified. This file is opened without reference to the VOC file. The file must be a hashed UniVerse file or a directory (UniVerse types 1 and 19).

pathname specifies the relative or absolute pathname of the file to be opened. If the file exists, it is opened and the THEN statements are executed; the ELSE statements are ignored. If *pathname* evaluates to the null value, the OPENPATH statement fails, the program terminates, and a run-time error message appears.

If the file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

Use the TO clause to assign the file to a *file.variable*. All statements used to read, write, delete, or clear that file must refer to it by the assigned *file.variable* name. If you do not assign the file to a *file.variable*, an internal default file variable is used. File references that do not specify *file.variable* access the most recently opened default file. The file opened to the default file variable is assigned to the system variable @STDFIL.

The ON ERROR Clause

The ON ERROR clause is optional in the OPENPATH statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the OPENPATH statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.

- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The STATUS Function

You can use the STATUS function after an OPENPATH statement to find the cause of a file open failure (that is, for an OPENPATH statement in which the ELSE clause is used). The following values can be returned if the OPENPATH statement is unsuccessful:

Value	Description
–1	Filename not found in the VOC file.
–2[1]	Null filename or file. This error may also occur when you cannot open a file across UV/Net.
–3	Operating system access error that occurs when you do not have permission to access a UniVerse file in a directory. For example, this may occur when trying to access a type 1 or type 30 file.
–4[1]	Access error when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
–5	Read error detected by the operating system.
–6	Unable to lock file header.
–7	Invalid file revision or wrong byte-ordering for the platform.
–8[1]	Invalid part file information.
–9[1]	Invalid type 30 file information in a distributed file.
–10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked “inconsistent.”
–11	The file is a view, therefore it cannot be opened by a BASIC program.
–12	No SQL privileges to open the table.

Value	Description
–13[1]	Index problem.
–14	Cannot open NFS file.

1. A generic error that can occur for various reasons.

Example

The following example opens the file SUN.MEMBER. The pathname specifies the file.

```
OPENPATH '/user/members/SUN.MEMBER' ELSE ABORT
```

OPENSEQ statement

Syntax

OPENSEQ *filename*, *record.ID* TO *file.variable* [USING *dynamic.array*]
[ON ERROR *statements*] [LOCKED *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

OPENSEQ *pathname* TO *file.variable* [USING *dynamic.array*]
[ON ERROR *statements*] [LOCKED *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the OPENSEQ statement to open a file for sequential processing. All sequential file references in a BASIC program must be preceded by an OPENSEQ or OPENDEV statement for that file. Although you can open several files for sequential processing at the same point in the program, you must issue a separate OPENSEQ statement for each. See the READSEQ and WRITESEQ statements for more details on sequential processing.

Note: Performing multiple OPENSEQ operations on the same file results in creating only one update record lock. This single lock can be released by a CLOSESEQ or RELEASE statement.

The first syntax is used to open a record in a type 1 or type 19 file.

The second syntax specifies a pathname to open a UNIX or DOS file. The file can be a disk file, a pipe, or a special device.

filename specifies the name of the type 1 or type 19 file containing the record to be opened.

record.ID specifies the record in the file to be opened. If the record exists and is not locked, the file is opened and the THEN statements are executed; the ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next statement. If the record or the file itself cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

pathname is an explicit pathname for the file, pipe, or device to be opened. If the file exists and is not locked, it is opened and the THEN statements are executed; the

ELSE statements are ignored. If the pathname does not exist, the ELSE statements are executed; any THEN statements are ignored.

If the file does not exist, the OPENSEQ statement fails. The file can also be explicitly created with the CREATE statement.

OPENSEQ sets an update record lock on the specified record or file. This lock is reset by a CLOSESEQ statement. This prevents any other program from changing the record while you are processing it.

If *filename*, *record.ID*, or *pathname* evaluate to the null value, the OPENSEQ statement fails, the program terminates, and a run-time error message appears.

The TO clause is required. It assigns the record, file, or device to *file.variable*. All statements used to sequentially read, write, delete, or clear that file must refer to it by the assigned file variable name.

File Buffering

Normally UniVerse uses buffering for sequential input and output operations. Use the NOBUF statement after an OPENSEQ statement to turn off buffering and cause all writes to the file to be performed immediately. For more information about file buffering, see the NOBUF statement.

The USING Clause

You can optionally include the USING clause to control whether the opened file is included in the rotating file pool. The USING clause supplements OPENSEQ processing with a dynamic array whose structure emulates an &DEVICE& file record. Field 17 of the dynamic array controls inclusion in the rotating file pool with the following values:

- Y removes the opened file.
- N includes the opened file.

The ON ERROR Clause

The ON ERROR clause is optional in the OPENSEQ statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the OPENSEQ statement is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED Clause

The LOCKED clause is optional, but recommended. Its syntax is the same as that of the ELSE clause. The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the OPENSEQ statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file
- Intent file
- Shared file
- Update record
- Shared record

If the OPENSEQ statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

Use the STATUS function after an OPENSEQ statement to determine whether the file was successfully opened.

The STATUS Function

The file type is returned if the file is opened successfully. If the file is not opened successfully, the following values may return:

Value	Description
–1	Filename not found in the VOC file.
–2[1]	Null filename or file. This error may also occur when you cannot open a file across UV/Net.
–3	Operating system access error that occurs when you do not have privileges to access a UniVerse file in a directory. For example, this may occur when trying to access a type 1 or type 30 file.
–4[1]	Access error when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
–5	Read error detected by the operating system.
–6	Unable to lock file header.
–7	Invalid file revision or wrong byte-ordering for the platform.
–8[1]	Invalid part file information.
–9[1]	Invalid type 30 file information in a distributed file.
–10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked “inconsistent.”
–11	The file is a view, therefore it cannot be opened by a BASIC program.
–12	No SQL privileges to open the table.
–13[1]	Index problem.
–14	Cannot open NFS file.

1. A generic error that can occur for various reasons.

Examples

The following example reads RECORD1 from the nonhashed file FILE.E:

```
OPENSEQ 'FILE.E', 'RECORD1' TO FILE THEN
   PRINT "'FILE.E' OPENED FOR PROCESSING"
   END ELSE ABORT
READSEQ A FROM FILE THEN PRINT A ELSE STOP
```

The next example writes the record read from FILE.E to the file */usr/depta/file1*:

```
OPENSEQ '/usr/depta/file1' TO OUTPUT THEN
   PRINT "usr/depta/file1 OPENED FOR PROCESSING"
END ELSE ABORT
WRITESEQ A ON OUTPUT ELSE PRINT "CANNOT WRITE TO OUTPUT"
    .
    .
    .
CLOSESEQ FILE
CLOSESEQ OUTPUT
END
```

This is the program output:

```
FILE.E OPENED FOR PROCESSING
HI THERE
    .
    .
    .
/usr/depta/file1 OPENED FOR PROCESSING
```

The next example includes the USING clause to remove an opened file from the rotating file pool:

```
DEVREC = "1"@FM
FOR I = 2 TO 16
   DEVREC = DEVREC:I:@FM
NEXT I
DEVREC=DEVREC:'Y'
*
OPENSEQ 'SEQTEST', 'TESTDATA' TO TESTFILE USING DEVREC
THEN PRINT "OPENED 'TESTDATA' OK...."
ELSE PRINT "COULD NOT OPEN TESTDATA"
CLOSESEQ TESTFILE
```

This is the program output:

```
OPENED 'TESTDATA' OK
```

Syntax

ORS (*array1*, *array2*)

CALL –ORS (*return.array*, *array1*, *array2*)

CALL !ORS (*return.array*, *array1*, *array2*)

Description

Use the ORS function to create a dynamic array of the logical OR of corresponding elements of two dynamic arrays.

Each element of the new dynamic array is the logical OR of the corresponding elements of *array1* and *array2*. If an element of one dynamic array has no corresponding element in the other dynamic array, a false is assumed for the missing element.

If both corresponding elements of *array1* and *array2* are the null value, null is returned for those elements. If one element is the null value and the other is 0 or an empty string, null is returned. If one element is the null value and the other is any value other than 0 or an empty string, a true is returned.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A="A":@SM:0:@VM:4:@SM:1
B=0:@SM:1-1:@VM:2
PRINT ORS(A,B)
```

This is the program output:

```
1s0v1s1
```

PAGE statement

Syntax

PAGE [ON *print.channel*] [*page#*]

Description

Use the PAGE statement to print headings, footings, and page advances at the appropriate places on the specified output device. You can specify headings and footings before execution of the PAGE statement (see the HEADING and FOOTING statements). If there is no heading or footing, PAGE clears the screen.

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement). Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

page# is an expression that specifies the next page number. If a heading or footing is in effect when the page number is specified, the heading or footing on the current page contains a page number equal to one less than the value of *page#*.

If either *print.channel* or *page#* evaluates to the null value, the PAGE statement fails, the program terminates, and a run-time error message appears.

Example

In the following example the current value of X provides the next page number:

```
PAGE ON 5 X
```

Syntax

PERFORM *command*

Description

Use the PERFORM statement to execute a UniVerse sentence, paragraph, menu, or command from within the BASIC program, then return execution to the statement following the PERFORM statement. The commands are executed in the same environment as the BASIC program that called them; that is, unnamed common variables, @variables, and in-line prompts retain their values, and select lists and the DATA stack remain active. If these values change, the new values are passed back to the calling program.

You can specify multiple commands in the PERFORM statement in the same way you specify them in the body of a UniVerse paragraph. Each command or line must be separated by a field mark (ASCII CHAR(254)).

If *command* evaluates to the null value, the PERFORM statement fails, the program terminates, and a run-time error message appears.

You cannot use the PERFORM statement within a transaction to execute most UniVerse commands and SQL statements. However, you can use PERFORM to execute the following UniVerse commands and SQL statements within a transaction:

UniVerse Commands and SQL Statements Used with PERFORM

CHECK.SUM	INSERT	SEARCH	SSELECT
COUNT	LIST	SELECT (RetrieVe)	STAT
DELETE (SQL)	LIST.ITEM	SELECT (SQL)	SUM
DISPLAY	LIST.LABEL	SORT	UPDATE
ESEARCH	RUN	SORT.ITEM	
GET.LIST	SAVE.LIST	SORT.LABEL	

REALITY Flavor

In a REALITY flavor account PERFORM can take all the clauses of the EXECUTE statement. To get these PERFORM characteristics in other flavor accounts, use the PERF.EQ.EXEC option of the $OPTIONS statement.

Example

In the following example multiple commands are separated by field marks:

```
PERFORM 'RUN BP SUB'
FM=CHAR(254)
COMMAND = 'SSELECT EM':FM
COMMAND := 'RUN BP PAY':FM
COMMAND := 'DATA 01/10/85'

PERFORM COMMAND
A = 'SORT EM '
A := 'WITH PAY.CODE EQ'
A := '10 AND WITH DEPT'
A := 'EQ 45'
PERFORM A
```

Syntax

PRECISION *expression*

Description

Use the PRECISION statement to control the maximum number of decimal places that are output when the system converts a numeric value from internal binary format to an ASCII character string value.

expression specifies a number from 0 through 9. Any fractional digits in the result of such a conversion that exceed the precision setting are rounded off.

If you do not include a PRECISION statement, a default precision of 4 is assumed. Precisions are stacked so that a BASIC program can change its precision and call a subroutine whose precision is the default unless the subroutine executes a PRECISION statement. When the subroutine returns to the calling program, the calling program has the same precision it had when it called the subroutine.

Trailing fractional zeros are dropped during output. Therefore, when an internal number is converted to an ASCII string, the result might appear to have fewer decimal places than the precision setting allows. However, regardless of the precision setting, the calculation always reflects the maximum accuracy of which the computer is capable (that is, slightly more than 17 total digits, including integers).

If *expression* evaluates to the null value, the PRECISION statement fails, the program terminates, and a run-time error message appears.

Example

```
A = 12.123456789
PRECISION 8
PRINT A
PRECISION 4
PRINT A
```

This is the program output:

```
12.12345679
12.1235
```

PRINT statement

Syntax

PRINT [ON *print.channel*] [*print.list*]

Description

Use the PRINT statement to send data to the screen, a line printer, or another print file.

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement). If *print.channel* evaluates to the null value, the PRINT statement fails, the program terminates, and a run-time error message appears. Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

You can specify HEADING, FOOTING, PAGE, and PRINTER CLOSE statements for each logical print channel. The contents of the print files are printed in order by logical print channel number.

print.list can contain any BASIC expression. The elements of the list can be numeric or character strings, variables, constants, or literal strings; the null value, however, cannot be printed. The list can consist of a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no *print.list* is designated, a blank line is printed.

Expressions separated by commas are printed at preset tab positions. The default tabstop setting is 10 characters. For information about changing the default setting, see the TABSTOP statement. Use multiple commas together for multiple tabulations between expressions.

Expressions separated by colons are concatenated. That is, the expression following the colon is printed immediately after the expression preceding the colon. To print a list without a LINEFEED and RETURN, end *print.list* with a colon (:).

Examples

```
A=25;B=30
C="ABCDE"
PRINT A+B
PRINT
```

```
PRINT "ALPHA ":C
PRINT "DATE ":PRINT "10/11/93"
*
PRINT ON 1 "FILE 1"
* The string "FILE 1" is printed on print file 1.
```

This is the program output:

```
55
ALPHA ABCDE
DATE 10/11/93
```

The following example clears the screen:

```
PRINT @(-1)
```

The following example prints the letter X at location column 10, row 5:

```
PRINT @(10,5) 'X'
```

PRINTER statement

Syntax

PRINTER { ON | OFF | RESET }

PRINTER CLOSE [ON *print.channel*]

Description

Use the PRINTER statement to direct output either to the screen or to a printer. By default, all output is sent to the screen unless a PRINTER ON is executed or the P option to the RUN command is used. See the SETPTR command for more details about redirecting output.

PRINTER ON sends output to the system line printer via print channel 0. The output is stored in a buffer until a PRINTER CLOSE statement is executed or the program terminates; the output is then printed (see the PRINTER CLOSE statement).

PRINTER OFF sends output to the screen via print channel 0. When the program is executed, the data is immediately printed on the screen.

The PRINTER ON or PRINTER OFF statement must precede the PRINT statement that starts the print file.

Use the PRINTER RESET statement to reset the printing options. PRINTER RESET removes the header and footer, resets the page count to 1, resets the line count to 1, and restarts page waiting.

Note: Use TPRINT to set a delay before printing. See also the TPARM statement.

The PRINTER CLOSE Statement

Use the PRINTER CLOSE statement to print all output data stored in the printer buffer.

You can specify print channel –1 through 255 with the ON clause. If you omit the ON clause from a PRINTER CLOSE statement, print channel 0 is closed. Only data directed to the printer specified by the ON clause is printed. Therefore, there must be a corresponding PRINTER CLOSE ON *print.channel* for each ON clause specified in a PRINT statement. All print channels are closed when the program stops. Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

If *print.channel* evaluates to the null value, the PRINTER CLOSE statement fails, the program terminates, and a run-time error message appears.

In PICK, IN2, and REALITY flavor accounts, the PRINTER CLOSE statement closes all print channels.

Example

```
PRINTER ON
PRINT "OUTPUT IS PRINTED ON PRINT FILE 0"
PRINTER OFF
PRINT "OUTPUT IS PRINTED ON THE TERMINAL"
*
PRINT ON 1 "OUTPUT WILL BE PRINTED ON PRINT FILE 1"
PRINT ON 2 "OUTPUT WILL BE PRINTED ON PRINT FILE 2"
```

This is the program output:

```
OUTPUT IS PRINTED ON THE TERMINAL
```

PRINTERR statement

Syntax

PRINTERR [*error.message*]

Description

Use the PRINTERR statement to print a formatted error message on the bottom line of the terminal. The message is cleared by the next INPUT statement or is overwritten by the next PRINTERR or INPUTERR statement. PRINTERR clears the type-ahead buffer.

error.message is an expression that evaluates to the error message text. The elements of the expression can be numeric or character strings, variables, constants, or literal strings. The null value cannot be an element because it cannot be output. The expression can be a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no error message is designated, a blank line is printed. If *error.message* evaluates to the null value, the default message is printed:

```
Message ID is NULL: undefined error
```

Expressions separated by commas are printed at preset tab positions. The default tabstop setting is 10 characters. For information about changing the default setting, see the TABSTOP statement. Use multiple commas together to cause multiple tabulations between expressions.

Expressions separated by colons are concatenated: that is, the expression following the colon is printed immediately after the expression preceding the colon.

See also the INPUTERR statement in Volume 1.

REALITY Flavor

In a REALITY flavor account the PRINTERR statement prints a formatted error message from the ERRMSG file on the bottom line of the terminal. REALITY syntax is:

PRINTERR [*dynamic.array*] [FROM *file.variable*]

dynamic.array must contain a record ID and any arguments to the message, with each element separated from the next by a field mark. If *dynamic.array* does not specify an existing record ID, a warning message states that no error message can be found.

If *dynamic.array* evaluates to the null value, the default error message is printed:

```
Message ID is NULL: undefined error
```

The FROM clause lets you read the error message from an open file. If *file.variable* evaluates to the null value, the PRINTERR statement fails, the program terminates, and a run-time error message appears.

This statement is similar to the STOP statement on a Pick system except that it does not terminate the program upon execution. You can use it wherever you can use a STOP or ABORT statement.

To use the REALITY version of the PRINTERR statement in PICK, IN2, INFORMATION, and IDEAL flavor accounts, use the USE.ERRMSG option of the $OPTIONS statement.

UniVerse provides a standard Pick ERRMSG file. You can construct a local ERRMSG file using the following syntax in the records. Each field must start with one of these codes, as shown in the following table:

ERRMSG File Codes

Code	Action
A[(*n*)]	Display next argument left-justified; *n* specifies field length.
D	Display system date.
E [*string*]	Display record ID of message in brackets; *string* displayed after ID.
H [*string*]	Display *string.*
L [(*n*)]	Output newline; *n* specifies number of newlines.
R [(*n*)]	Display next argument right-justified; *n* specifies field length.
S [(*n*)]	Output *n* blank spaces from beginning of line.
T	Display system time.

PROCREAD statement

Syntax

PROCREAD *variable*
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the PROCREAD statement to assign the contents of the primary input buffer to a variable. Your BASIC program must be called by a proc. If your program was not called from a proc, the ELSE statements are executed; otherwise the THEN statements are executed.

If *variable* evaluates to the null value, the PROCREAD statement fails, the program terminates, and a run-time error message appears.

Syntax

PROCWRITE *string*

Description

Use the PROCWRITE statement to write *string* to the primary input buffer. Your program must be called by a proc.

If *string* evaluates to the null value, the PROCWRITE statement fails, the program terminates, and a run-time error message appears.

PROGRAM statement

Syntax

PROG[RAM] [*name*]

Description

Use the PROGRAM statement to identify a program. The PROGRAM statement is optional; if you use it, it must be the first noncomment line in the program.

name can be specified for documentation purposes; it need not be the same as the actual program name.

Example

```
PROGRAM BYSTATE
```

Syntax

PROMPT *character*

Description

Use the PROMPT statement to specify the character to be displayed on the screen when user input is required. If no PROMPT statement is issued, the default prompt character is the question mark (?).

If *character* evaluates to more than one character, only the first character is significant; all others are ignored.

The prompt character becomes *character* when the PROMPT statement is executed. Although the value of *character* can change throughout the program, the prompt character remains the same until a new PROMPT statement is issued or the program ends.

Generally, data the user enters in response to the prompt appears on the screen. If the source of the input is something other than the keyboard (for example, a DATA statement), the data is displayed on the screen after the prompt character. Use PROMPT " " to prevent any prompt from being displayed. PROMPT " " also suppresses the display of input from DATA statements.

If *character* evaluates to the null value, no prompt appears.

Examples

Source Lines	Program Output

`PROMPT "HELLO"`
`PRINT "ENTER ANSWER ":`
`INPUT ANS`
`PROMPT "-"`
`PRINT "ENTER ANSWER ":`
`INPUT ANS`
`PROMPT ""`
`PRINT "ENTER ANSWER ":`
`INPUT ANS`
`END`
	`ENTER ANSWER H`**`ANSWER`**
	`ENTER ANSWER -`**`YES`**
	`ENTER ANSWER `**`NO`**

PWR function

Syntax

PWR (*expression*, *power*)

Description

Use the PWR function to return the value of expression raised to the power specified by *power*.

The PWR function operates like exponentiation (that is, PWR(X,Y) is the same as X**Y).

A negative value cannot be raised to a noninteger power. If it is, the result of the function is PWR(–X,Y) and an error message is displayed.

If either *expression* or *power* is the null value, null is returned.

On overflow or underflow, a warning is printed and 0 is returned.

Example

```
A=3
B=PWR(5,A)
PRINT "B= ",B
```

This is the program output:

```
B=    125
```

QUOTE function

Syntax

QUOTE (*expression*)

Description

Use the QUOTE function to enclose an expression in double quotation marks. If *expression* evaluates to the null value, null is returned, without quotation marks.

Example

```
PRINT QUOTE(12 + 5) : " IS THE ANSWER."
END
```

This is the program output:

```
"17" IS THE ANSWER.
```

RAISE function

Syntax

RAISE (*expression*)

Description

Use the RAISE function to return a value equal to *expression*, except that system delimiters in *expression* are converted to the next higher-level delimiter: value marks are changed to field marks, subvalue marks are changed to value marks, and so on. If *expression* evaluates to the null value, null is returned.

The conversions are:

IM	CHAR(255)	to	IM	CHAR(255)
FM	CHAR(254)	to	IM	CHAR(255)
VM	CHAR(253)	to	FM	CHAR(254)
SM	CHAR(252)	to	VM	CHAR(253)
TM	CHAR(251)	to	SM	CHAR(252)
	CHAR(250)	to		CHAR(251)
	CHAR(249)	to		CHAR(250)
	CHAR(248)	to		CHAR(249)

The PIOPEN Flavor

In PIOPEN flavor, the delimiters that can be raised are CHAR(255) through CHAR(252). All other characters are left unchanged. You can obtain PIOPEN flavor for the RAISE function by:

- Compiling your program in a PIOPEN flavor account
- Specifying the $OPTIONS INFO.MARKS statement

Examples

In the following examples an item mark is shown by `I`, a field mark is shown by `F`, a value mark is shown by `V`, and a subvalue mark is shown by `S`.

The following example sets A to `DDIEEI123I777`:

```
A= RAISE('DD':FM'EE':FM:123:FM:777)
```

The next example sets B to `1I2F3I4V5`:

```
B= RAISE(1:IM:2:VM:3:FM:4:SM:5)
```

The next example sets C to `999S888`:

```
C= RAISE(999:TM:888)
```

RANDOMIZE statement

Syntax

RANDOMIZE (*expression*)

Description

Use the RANDOMIZE statement with an expression to make the RND function generate the same sequence of random numbers each time the program is run. If no expression is supplied, or if *expression* evaluates to the null value, the internal time of day is used (the null value is ignored). In these cases the sequence is different each time the program is run.

Example

```
RANDOMIZE (0)
FOR N=1 TO 10
   PRINT RND(4):' ':
NEXT N
PRINT
*
RANDOMIZE (0)
FOR N=1 TO 10
   PRINT RND(4):' ':
NEXT
PRINT
*
RANDOMIZE (3)
FOR N=1 TO 10
   PRINT RND(4):' ':
NEXT N
PRINT
```

This is the program output:

```
0 2 1 2 0 2 1 2 1 1
0 2 1 2 0 2 1 2 1 1
2 0 1 1 2 1 0 1 2 3
```

Syntax

READ *dynamic.array* FROM [*file.variable,*] *record.ID* [ON ERROR *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

{READL | READU} *dynamic.array* FROM [*file.variable,*] *record.ID*
[ON ERROR *statements*] [LOCKED *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

READV *dynamic.array* FROM [*file.variable,*] *record.ID, field#*
[ON ERROR *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

{READVL | READVU} *dynamic.array* FROM [*file.variable,*] *record.ID, field#*
[ON ERROR *statements*] [LOCKED *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use READ statements to assign the contents of a record from a UniVerse file to *dynamic.array.*

Uses for READ Statements

Use this statement...	To do this...
READ	Read a record.
READL	Acquire a shared record lock and read a record.
READU	Acquire an update record lock and read a record.
READV	Read a field.
READVL	Acquire a shared record lock and read a field.
READVU	Acquire an update record lock and read a field.

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement). If the file is neither accessible nor open, the program terminates and a run-time error message appears.

If *record.ID* exists on the specified file, *dynamic.array* is set to the contents of the record, and the THEN statements are executed; any ELSE statements are ignored. If no THEN statements are specified, program execution continues with the next statement. If *record.ID* does not exist, *dynamic.array* is set to an empty string, and the ELSE statements are executed; any THEN statements are ignored.

If *file.variable*, *record.ID*, or *field#* evaluate to the null value, the READ statement fails, the program terminates, and a run-time error message appears.

If the file is an SQL table, the effective user of the program must have SQL SELECT privilege to READ records in the file. See the AUTHORIZATION statement in Volume 1 for information about the effective user of a program.

If the file is a distributed file, you can use the STATUS function after a READ statement is executed, to determine the results of the operation, as follows:

–1 The partitioning algorithm does not evaluate to an integer.

–2 The part number is invalid.

READL and READU Statements

Use the READL syntax to acquire a shared record lock and then read the record. This allows other programs to read the record with no lock or a shared record lock.

Use the READU statement to acquire an update record lock and then read the record. The update record lock prevents other users from updating the record until the user who owns it releases it.

An update record lock can only be acquired when no shared record lock exists. It can be promoted from a shared record lock owned by the user requesting the update record lock if no shared record locks exist.

To prevent more than one program or user from modifying the same record at the same time, use READU instead of READ.

READV, READVL, and READVU Statements

Use the READV statement to assign the contents of a field in a UniVerse file record to *dynamic.array*.

Use the READVL statement to acquire a shared record lock and then read a field from the record. The READVL statement conforms to all the specifications of the READL and READV statements.

Use the READVU statement to acquire an update record lock and then read a field from the record. The READVU statement conforms to all the specifications of the READU and READV statements.

You can specify *field#* only with the READV, READVL, and READVU statements. It specifies the index number of the field to be read from the record. You can use a *field#* of 0 to determine whether the record exists. If the field does not exist, *dynamic.array* is assigned the value of an empty string.

The ON ERROR Clause

The ON ERROR clause is optional in the READ statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the READ statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED Clause

You can use the LOCKED clause only with the READL, READU, READVL, and READVU statements. Its syntax is the same as that of the ELSE clause.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the READ statement from being processed. The LOCKED clause is executed if one of the following conflicting locks exists:

In this statement...	This requested lock...	Conflicts with these locks...
READL READVL	Shared record	Exclusive file Update record
READU READVU	Update record	Exclusive file Intent file Shared file Update record Shared record

If a READ statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

Releasing Locks

A shared record lock can be released with a CLOSE, RELEASE, or STOP statement. An update record lock can be released with a CLOSE, DELETE, MATWRITE, RELEASE, STOP, WRITE, or WRITEV statement.

Locks acquired or promoted within a transaction are not released when the previous statements are processed.

All record locks are released when you return to the UniVerse system prompt.

PICK, IN2, and REALITY Flavors

In PICK, IN2, and REALITY flavor accounts, if *record.ID* or *field#* does not exist, *dynamic.array* retains its value and is not set to an empty string. The ELSE statements are executed; any THEN statements are ignored. To specify PICK, IN2, and REALITY flavor READ statements in an INFORMATION or IDEAL flavor account, use the READ.RETAIN option of the $OPTIONS statement.

Examples

```
OPEN '','SUN.MEMBER' TO FILE ELSE STOP
FOR ID=5000 TO 6000
   READ MEMBER FROM FILE, ID THEN PRINT ID ELSE NULL
NEXT ID
```

```
OPEN '','SUN.SPORT' ELSE STOP 'CANT OPEN "SUN.SPORT"'
READ ID FROM "853333" ELSE
   PRINT 'CANT READ ID "853333" ON FILE "SUN.SPORT"'
END
```

```
X="6100"
READ PERSON FROM FILE,X THEN PRINT PERSON<1> ELSE
   PRINT "PERSON ":X:" NOT ON FILE"
END
```

The next example locks the record N in the file SUN.MEMBER, reads field 3 (STREET) from it, and prints the value of the field:

```
OPEN '','SUN.MEMBER' TO FILE ELSE STOP
FOR N=5000 TO 6000
   READVU STREET FROM FILE,N,3 THEN PRINT STREET ELSE NULL
   RELEASE
NEXT
```

```
OPEN "DICT","MYFILE" TO DICT.FILE ELSE STOP
OPEN "","MYFILE" ELSE STOP ; *USING DEFAULT FILE VARIABLE
READU ID.ITEM FROM DICT.FILE,"@ID" ELSE
   PRINT "NO @ID"
   STOP
END
```

This is the program output:

```
5205
5390
CANT READ ID "853333" ON FILE "SUN.SPORT"
MASTERS
4646 TREMAIN DRIVE
670 MAIN STREET
```

READBLK statement

Syntax

READBLK *variable* FROM *file.variable*, *blocksize*
{ THEN *statements* [ELSE *statements*] | ELSE *statements* }

Description

Use the READBLK statement to read a block of data of a specified length from a file opened for sequential processing and assign it to a variable. The READBLK statement reads a block of data beginning at the current position in the file and continuing for *blocksize* bytes and assigns it to *variable*. The current position is reset to just beyond the last byte read.

file.variable specifies a file previously opened for sequential processing.

If the data can be read from the file, the THEN statements are executed; any ELSE statements are ignored. If the file is not readable or if the end of file is encountered, the ELSE statements are executed and the THEN statements are ignored. If the ELSE statements are executed, *variable* is set to an empty string.

If either *file.variable* or *blocksize* evaluates to the null value, the READBLK statement fails, the program terminates, and a run-time error message appears.

Note: A newline in UNIX files is one byte long, whereas in Windows NT it is two bytes long. This means that for a file with newlines, the same READBLK statement may return a different set of data depending on the operating system the file is stored under.

In the event of a timeout, READBLK returns no bytes from the buffer, and the entire I/O operation must be retried.

The difference between the READSEQ statement and the READBLK statement is that the READBLK statement reads a block of data of a specified length, whereas the READSEQ statement reads a single line of data.

On Windows NT systems, if you use READBLK to read data from a 1/4-inch cartridge drive (60 or 150 MB) that you open with the OPENDEV statement, you must use a block size of 512 bytes or a multiple of 512 bytes.

For more information about sequential file processing, see the OPENSEQ, READSEQ, and WRITESEQ statements.

Example

```
OPENSEQ 'FILE.E', 'RECORD4' TO FILE ELSE ABORT
READBLK VAR1 FROM FILE, 15 THEN PRINT VAR1
PRINT
READBLK VAR2 FROM FILE, 15 THEN PRINT VAR2
```

This is the program output:

```
FIRST LINE
SECO

ND LINE
THIRD L
```

READL statement

Use the READL statement to acquire a shared record lock and perform the READ statement. For details, see the READ statement.

Syntax

READLIST *dynamic.array* [FROM *list.number*]
{ THEN *statements* [ELSE *statements*] | ELSE *statements* }

Description

Use the READLIST statement to read the remainder of an active select list into a dynamic array.

list.number is an expression that evaluates to the number of the select list to be read. It can be from 0 through 10. If you do not use the FROM clause, select list 0 is used.

READLIST reads all elements in the active select list. If a READNEXT statement is used on the select list before the READLIST statement, only the elements not read by the READNEXT statement are stored in *dynamic.array*. READLIST empties the select list.

If one or more elements are read from *list.number*, the THEN statements are executed. If there are no more elements in the select list or if a select list is not active, the ELSE statements are executed; any THEN statements are ignored.

If *list.number* evaluates to the null value, the READLIST statement fails, the program terminates, and run-time error message appears.

In IDEAL and INFORMATION flavor accounts, use the VAR.SELECT option of the $OPTIONS statement to get READLIST to behave as it does in PICK flavor accounts.

PICK, REALITY, and IN2 Flavors

In PICK, REALITY, and IN2 flavor accounts, the READLIST statement has the following syntax:

READLIST *dynamic.array* FROM *listname* [SETTING *variable*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

In these flavors the READLIST statement reads a saved select list from the &SAVEDLISTS& file without activating a select list. In PICK and IN2 flavor accounts, READLIST lets you access a saved select list without changing the currently active select list if there is one.

The select list saved in *listname* in the &SAVEDLISTS& file is put in *dynamic.array*. The elements of the list are separated by field marks.

listname can be of the form

record.ID

or

record.ID account.name

record.ID specifies the record ID of the list in &SAVEDLISTS&, and *account.name* specifies the name of another UniVerse account in which to look for the &SAVEDLISTS& file.

The SETTING clause assigns the count of the elements in the list to *variable*.

If the list is retrieved successfully (the list must not be empty), the THEN statements are executed; if not, the ELSE statements are executed. If *listname* evaluates to the null value, the READLIST statement fails, the program terminates, and a run-time error message appears.

In PICK, REALITY, and IN2 flavor accounts, use the –VAR.SELECT option of the $OPTIONS statement to get READLIST to behave as it does in IDEAL flavor accounts.

Syntax

READNEXT *dynamic.array* [,*value* [,*subvalue*]] [FROM *list*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the READNEXT statement to assign the next record ID from an active select list to *dynamic.array*.

list specifies the select list. If none is specified, select list 0 is used. *list* can be a number from 0 through 10 indicating a numbered select list, or the name of a select list variable.

The BASIC SELECT statement or the UniVerse GET.LIST, FORM.LIST, SELECT, or SSELECT verbs create an active select list; these commands build the list of record IDs. The READNEXT statement reads the next record ID on the list specified in the FROM clause and assigns it to the *dynamic.array*.

When the select list is exhausted, *dynamic.array* is set to an empty string, and the ELSE statements are executed; any THEN statements are ignored.

If *list* evaluates to the null value, the READNEXT statement fails, the program terminates, and a run-time error message appears.

A READNEXT statement with *value* and *subvalue* specified accesses an exploded select list. The record ID is stored in *dynamic.array*, the value number in *value*, and the subvalue number in *subvalue*. If only *dynamic.array* is specified, it is set to a multivalued field consisting of the record ID, value number, and subvalue number, separated by value marks.

INFORMATION Flavor

In INFORMATION flavor accounts READNEXT returns an exploded select list. Use the RNEXT.EXPL option of the $OPTIONS statement to return exploded select lists in other flavors.

Example

```
OPEN '','SUN.MEMBER' ELSE STOP "CAN'T OPEN FILE"
SELECT TO 1
10: READNEXT MEM FROM 1 THEN PRINT MEM ELSE GOTO 15:
GOTO 10:
```

```
*
15: PRINT

OPEN '','SUN.SPORT' TO FILE ELSE STOP
SELECT FILE
COUNT=0
20*
READNEXT ID ELSE
PRINT 'COUNT= ',COUNT
STOP
END
COUNT=COUNT+1
GOTO 20
```

This is the program output:

```
4108
6100
3452
5390
7100
4500
2430
2342
6783
5205
4439
6203
7505
4309
1111
COUNT= 14
```

Syntax

READSEQ *variable* FROM *file.variable* [ON ERROR *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the READSEQ statement to read a line of data from a file opened for sequential processing. Sequential processing lets you process data one line at a time. UniVerse keeps a pointer at the current position in the file. The OPENSEQ statement sets this pointer to the first byte of the file, and it is advanced by READSEQ, READBLK, WRITESEQ, and WRITEBLK statements.

Each READSEQ statement reads data from the current position in the file up to a newline and assigns it to *variable*. The pointer is then set to the position following the newline. The newline is discarded.

file.variable specifies a file previously opened for sequential processing. The FROM clause is required. If the file is neither accessible nor open, or if *file.variable* evaluates to the null value, the READSEQ statement fails, the program terminates, and a run-time error message appears.

If data is read from the file, the THEN statements are executed, and the ELSE statements are ignored. If the file is not readable, or the end of file is encountered, the ELSE statements are executed; any THEN statements are ignored.

In the event of a timeout, READSEQ returns no bytes from the buffer, and the entire I/O operation must be retried.

READSEQ affects the STATUS function in the following way:

0 The read is successful.

1 The end of file is encountered.

2 A timeout ended the read.

–1 The file is not open for a read.

The ON ERROR Clause

The ON ERROR clause is optional in the READSEQ statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the READSEQ statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

Example

```
OPENSEQ 'FILE.E', 'RECORD4' TO FILE ELSE ABORT
FOR N=1 TO 3
   READSEQ A FROM FILE THEN PRINT A
NEXT N
CLOSESEQ FILE
```

This is the program output:

```
FIRST LINE
SECOND LINE
THIRD LINE
```

Syntax

READT [UNIT (*mtu*)] *variable*
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the READT statement to read the next tape record from a magnetic tape unit and assign its contents to a variable.

The UNIT clause specifies the number of the tape drive unit. Tape unit 0 is used if no unit is specified.

mtu is an expression that evaluates to a code made up of three decimal digits, as shown in the following table:

***mtu* Codes**

Code	Available Options
m (mode)	0 = No conversion 1 = EBCDIC conversion 2 = Invert high bit 3 = Invert high bit and EBCDIC conversion
t (tracks)	0 = 9 tracks. Only 9-track tapes are supported.
u (unit number)	0 through 7

The *mtu* expression is read from right to left. Therefore, if *mtu* evaluates to a one-digit code, it represents the tape unit number. If *mtu* evaluates to a two-digit code, the rightmost digit represents the unit number and the digit to its left is the track number; and so on.

If either *mtu* or *variable* evaluates to the null value, the READT statement fails, the program terminates, and a run-time error message appears.

Each tape record is read and processed completely before the next record is read. The program waits for the completion of data transfer from the tape before continuing.

If the next tape record exists, *variable* is set to the contents of the record, and the THEN statements are executed. If no THEN statements are specified, program execution continues with the next statement.

READT statement

Before a READT statement is executed, a tape drive unit must be attached (assigned) to the user. Use the ASSIGN command to assign a tape unit to a user. If no tape unit is attached or if the unit specification is incorrect, the ELSE statements are executed and the value assigned to *variable* is empty. Any THEN statements are ignored.

The largest tape record that the READT statement can read is system-dependent. If a tape record is larger than the system maximum, only the bytes up to the maximum are assigned to *variable*.

You can use the STATUS function after a READT statement is executed to determine the result of the operation. The value returned by the STATUS function for the READT statement is hardware-dependent (that is, it varies according to the characteristics of the specific tape drive). Consult the documentation that accompanied your tape drive for information on interpreting the values returned by the STATUS function.

PIOPEN Flavor

If you have a program that specifies the syntax UNIT *ndmtu*, the *nd* elements are ignored by the compiler and no errors are reported.

Examples

The following example reads a tape record from tape drive 0:

```
READT RECORD ELSE PRINT "COULD NOT READ FROM TAPE"
```

The next example reads a record from tape drive 3, doing an EBCDIC conversion in the process:

```
READT UNIT(103) RECORD ELSE PRINT "COULD NOT READ"
```

READU statement

Use the READU statement to set an update record lock and perform the READ statement. For details, see the READ statement.

READV statement

Use the READV statement to read the contents of a specified field of a record in a UniVerse file. For details, see the READ statement.

READVL statement

Use the READVL statement to set a shared record lock and perform the READV statement. For details, see the READ statement.

READVU statement

Use the READVU statement to set an update record lock and read the contents of a specified field of a record in a UniVerse file. For details, see the READ statement.

REAL function

Syntax

REAL (*number*)

Description

Use the REAL function to convert *number* into a floating-point number without loss of accuracy. If *number* evaluates to the null value, null is returned.

RECORDLOCKED function

Syntax

RECORDLOCKED (*file.variable*, *record.ID*)

Description

Use the RECORDLOCKED function to return the status of a record lock.

file.variable is a file variable from a previous OPEN statement.

record.ID is an expression that evaluates to the record ID of the record that is to be checked.

An insert file of equate names is provided to let you use mnemonics (see the following table). The insert file is called RECORDLOCKED.INS.IBAS, and is located in the INCLUDE directory in the UV account directory. In PIOPEN flavor accounts, the VOC file has a file pointer called SYSCOM. SYSCOM references the INCLUDE directory in the UV account directory.

To use the insert file, specify $INCLUDE SYSCOM RECORDLOCKED.INS.IBAS when you compile the program.

RECORDLOCKED.INS.IBAS File Equate Names

Equate Name	Value	Meaning
LOCK$MY.FILELOCK	3	This user has a FILELOCK.
LOCK$MY.READL	2	This user has a shared record lock.
LOCK$MY.READU	1	This user has an update record lock.
LOCK$NO.LOCK	0	The record is not locked.
LOCK$OTHER.READL	–1	Another user has a shared record lock.
LOCK$OTHER.READU	–2	Another user has an update record lock.
LOCK$OTHER.FILELOCK	–3	Another user has a FILELOCK.

If you have locked the file, the RECORDLOCKED function indicates only that you have the file lock for that record. It does not indicate any update record or shared record lock that you also have on the record.

Value Returned by the STATUS Function

Possible values returned by the STATUS function, and their meanings, are as follows:

> 0 A positive value is the terminal number of the owner of the lock (or the first terminal number encountered, if more than one user has locked records in the specified file).

< 0 A negative value is –1 times the terminal number of the remote user who has locked the record or file.

Examples

The following program checks to see if there is an update record lock or FILELOCK held by the current user on the record. If the locks are not held by the user, the ELSE clause reminds the user that an update record lock or FILELOCK is required on the record.This example using the SYSCOM file pointer, only works in PI/open flavor accounts.

```
$INCLUDE SYSCOM RECORDLOCKED.INS.IBAS
OPEN '','EMPLOYEES' TO EMPLOYEES
 ELSE STOP 'CANNOT OPEN FILE'
 .
 .
 .
IF RECORDLOCKED(EMPLOYEES,RECORD.ID) >= LOCK$MY.READU THEN
 GOSUB PROCESS.THIS.RECORD:
ELSE PRINT 'Cannot process record : ':RECORD.ID :', READU or
FILELOCK required.'
```

The next program checks to see if the record lock is held by another user and prints a message where the STATUS function gives the terminal number of the user who holds the record lock:

```
$INCLUDE SYSCOM RECORDLOCKED.INS.IBAS
OPEN '','EMPLOYEES' TO EMPLOYEES
ELSE STOP 'CANNOT OPEN FILE'
 .
 .
 .
IF RECORDLOCKED(EMPLOYEES,RECORD.ID) < LOCK$NO.LOCK THEN
 PRINT 'Record locked by user' : STATUS()
END
```

RECORDLOCK statements

Syntax

RECORDLOCKL *file.variable*, *record.ID* [ON ERROR *statements*]
[LOCKED *statements*]

RECORDLOCKU *file.variable*, *record.ID* [ON ERROR *statements*]
[LOCKED *statements*]

Description

Use RECORDLOCK statements to acquire a record lock on a record without reading the record.

Use this statement...	To acquire this lock without reading the record...
RECORDLOCKL	Shared record
RECORDLOCKU	Update record

file.variable is a file variable from a previous OPEN statement.

record.ID is an expression that evaluates to the record ID of the record that is to be locked.

The RECORDLOCKL Statement

The RECORDLOCKL statement lets other users lock the record using RECORDLOCK or any other statement that sets a shared record lock, but cannot gain exclusive control over the record with FILELOCK, or any statement that sets an update record lock.

The RECORDLOCKU Statement

The RECORDLOCKU statement prevents other users from accessing the record using a FILELOCK statement or any statement that sets either a shared record lock or an update record lock. You can reread a record after you have locked it; you are not affected by your own locks.

The ON ERROR Clause

The ON ERROR clause is optional in RECORDLOCK statements. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while a RECORDLOCK statement is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED Clause

The LOCKED clause is optional, but recommended.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the RECORDLOCK statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

For this statement…	This requested lock…	Conflicts with these locks…
RECORDLOCKL	Shared record	Exclusive file Update record
RECORDLOCKU	Update record	Exclusive file Intent file Shared file Update record Shared record

If the RECORDLOCK statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

Releasing Locks

A shared record lock can be released with a CLOSE, RELEASE, or STOP statement.

An update record lock can be released by a CLOSE, DELETE, MATWRITE, RELEASE, STOP, WRITE or WRITEV statement.

Locks acquired or promoted within a transaction are not released when the previous statements are processed.

All record locks are released when you return to the UniVerse system prompt.

Example

In the following example, the file EMPLOYEES is opened. Record 23694 is locked. If the record was already locked, the program terminates, and an appropriate message is displayed. The RECORDLOCKL statement allows other users to read the record with READL or lock it with another RECORDLOCKL, but prevents any other user from gaining exclusive control over the record.

```
OPEN '','EMPLOYEES' TO EMPLOYEES ELSE STOP 'Cannot open file'
RECORDLOCKL EMPLOYEES,'23694'
 LOCKED STOP 'Record previously locked by user ':STATUS(
```

Syntax

RELEASE [*file.variable* [,*record.ID*]] [ON ERROR *statements*]

Description

Use the RELEASE statement to unlock, or release, locks set by a FILELOCK, MATREADL, MATREADU, READL, READU, READVL, READVU, and OPENSEQ statement. These statements lock designated records to prevent concurrent updating by other users. If you do not explicitly release a lock that you have set, it is unlocked automatically when the program terminates.

file.variable specifies an open file. If *file.variable* is not specified and a record ID is specified, the default file is assumed (for more information on default files, see the OPEN statement). If the file is neither accessible nor open, the program terminates and a run-time error message appears.

record.ID specifies the lock to be released. If it is not specified, all locks in the specified file (that is, either *file.variable* or the default file) are released. If either *file.variable* or *record.ID* evaluates to the null value, the RELEASE statement fails, the program terminates, and a run-time error message appears.

When no options are specified, all locks in all files set by any FILELOCK, READL, READU, READVL, READVU, WRITEU, WRITEVU, MATREADL, MATREADU, MATWRITEU, or OPENSEQ statements during the current login session are released.

A RELEASE statement within a transaction is ignored.

The ON ERROR Clause

The ON ERROR clause is optional in the RELEASE statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the RELEASE statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.

- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

Examples

The following example releases all locks set in all files by the current program:

```
RELEASE
```

The next example releases all locks set in the NAMES file:

```
RELEASE NAMES
```

The next example releases the lock set on the record QTY in the PARTS file:

```
RELEASE PARTS, "QTY"
```

Syntax

REM (*dividend*, *divisor*)

Description

Use the REM function to calculate the remainder after integer division is performed on the dividend expression by the divisor expression.

The REM function calculates the remainder using the following formula:

REM (X, Y) = X – (INT (X / Y) * Y)

dividend and *divisor* can evaluate to any numeric value, except that *divisor* cannot be 0. If *divisor* is 0, a division by 0 warning message is printed, and 0 is returned. If either *dividend* or *divisor* evaluates to the null value, null is returned.

The REM function works like the MOD function.

Example

```
X=85; Y=3
PRINT 'REM (X,Y)= ',REM (X,Y)
```

This is the program output:

```
REM (X,Y)=     1
```

REM statement

Syntax

REM [*comment.text*]

Description

Use the REM statement to insert a comment in a BASIC program. Comments explain or document various parts of a program. They are part of the source code only and are nonexecutable. They do not affect the size of the object code.

A comment must be a separate BASIC statement, and can appear anywhere in a program. A comment must be one of the following comment designators:

REM * ! $*

Any text that appears between a comment designator and the end of a physical line is treated as part of the comment. If a comment does not fit on one physical line, it can be continued on the next physical line only by starting the new line with a comment designator. If a comment appears at the end of a physical line containing an executable statement, you must treat it as if it were a new statement and put a semicolon (;) after the executable statement, before the comment designator.

Example

```
PRINT "HI THERE"; REM This part is a comment.
REM This is also a comment and does not print.
REM
IF 5<6 THEN PRINT "YES"; REM A comment; PRINT "PRINT ME"
REM BASIC thinks PRINT "PRINT ME" is also part
REM of the comment.
IF 5<6 THEN
   PRINT "YES"; REM Now it doesn't.
   PRINT "PRINT ME"
END
```

This is the program output:

```
HI THERE
YES
YES
PRINT ME
```

Syntax

REMOVE (*dynamic.array, variable*)

Description

Use the REMOVE function to successively extract and return dynamic array elements that are separated by system delimiters, and to indicate which system delimiter was found. When a system delimiter is encountered, the value of the extracted element is returned. The REMOVE function is more efficient than the EXTRACT function for extracting successive fields, values, and so on, for multi-value list processing.

dynamic.array is an expression that evaluates to the dynamic array from which to extract elements.

variable is set to a code corresponding to the system delimiter which terminates the extracted element. The contents of *variable* indicate which system delimiter was found, as follows:

0 End of string

1 Item mark ASCII CHAR(255)

2 Field mark ASCII CHAR(254)

3 Value mark ASCII CHAR(253)

4 Subvalue mark ASCII CHAR(252)

5 Text mark ASCII CHAR(251)

6 ASCII CHAR(250) - Not available in the PIOPEN flavor

7 ASCII CHAR(249) - Not available in the PIOPEN flavor

8 ASCII CHAR(248) - Not available in the PIOPEN flavor

The REMOVE function extracts one element each time it is executed, beginning with the first element in *dynamic.array*. The operation can be repeated until all elements of *dynamic.array* are extracted. The REMOVE function does not change the dynamic array.

As each successive element is extracted from *dynamic.array*, a pointer associated with *dynamic.array* is set to the beginning of the next element to be extracted. Thus the pointer is advanced every time the REMOVE function is executed.

The pointer is reset to the beginning of *dynamic.array* whenever *dynamic.array* is reassigned. Therefore, *dynamic.array* should not be assigned a new value until all elements have been extracted (that is, until *variable* is 0).

If *dynamic.array* evaluates to the null value, null is returned and *variable* is set to 0 (end of string). If an element in *dynamic.array* is the null value, null is returned for that element, and *variable* is set to the appropriate delimiter code.

Unlike the EXTRACT function, the REMOVE function maintains a pointer into the dynamic array. (The EXTRACT function always starts processing at the beginning of the dynamic array, counting field marks, value marks, and subvalue marks until it finds the correct element to extract.)

See the REMOVE statement for the statement equivalent of this function.

Examples

The first example sets the variable FIRST to the string MIKE and the variable X to 2 (field mark). The second example executes the REMOVE function and PRINT statement until all the elements have been extracted, at which point A = 0. Printed lines are 12, 4, 5, 7654, and 00.

Source Lines	Program Output

`FM=CHAR(254)`
`NAME='MIKE':FM:'JOHN':FM`
`X=REMOVE(NAME,FIRST)`
`PRINT 'FIRST = ':FIRST, 'X = ':X`
	`FIRST = 2  X = MIKE`

`VM=CHAR(253)`
`A = 1`
`Z=12:VM:4:VM:5:VM:7654:VM:00`
`FOR X=1 TO 20 UNTIL A=0`
`  A = REMOVE(Z,Y)`
`  PRINT 'Y = ':Y, 'A = ':A`
`NEXT X`
	`Y = 3  A = 12`
	`Y = 3  A = 4`
	`Y = 3  A = 5`
	`Y = 3  A = 7654`
	`Y = 0  A = 0`

Syntax

REMOVE *element* FROM *dynamic.array* SETTING *variable*

Description

Use the REMOVE statement to successively extract dynamic array elements that are separated by system delimiters. When a system delimiter is encountered, the extracted element is assigned to *element*. The REMOVE statement is more efficient than the EXTRACT function for extracting successive fields, values, and so on, for multivalue list processing.

dynamic.array is an expression that evaluates to the dynamic array from which to extract elements.

variable is set to a code value corresponding to the system delimiter terminating the element just extracted. The delimiter code settings assigned to *variable* are as follows:

0 End of string

1 Item mark ASCII CHAR(255)

2 Field mark ASCII CHAR(254)

3 Value mark ASCII CHAR(253)

4 Subvalue mark ASCII CHAR(252)

5 Text mark ASCII CHAR(251)

6 ASCII CHAR(250) - Not supported in the PIOPEN flavor

7 ASCII CHAR(249) - Not supported in the PIOPEN flavor

8 ASCII CHAR(248) - Not supported in the PIOPEN flavor

The REMOVE statement extracts one element each time it is executed, beginning with the first element in *dynamic.array*. The operation can be repeated until all elements of *dynamic.array* are extracted. The REMOVE statement does not change the dynamic array.

As each element is extracted from *dynamic.array* to *element*, a pointer associated with *dynamic.array* is set to the beginning of the next element to be extracted. Thus, the pointer is advanced every time the REMOVE statement is executed.

The pointer is reset to the beginning of *dynamic.array* whenever *dynamic.array* is reassigned. Therefore, *dynamic.array* should not be assigned a new value until all elements have been extracted (that is, until *variable* = 0).

If an element in *dynamic.array* is the null value, null is returned for that element.

Unlike the EXTRACT function, the REMOVE statement maintains a pointer into the dynamic array. (The EXTRACT function always starts processing at the beginning of the dynamic array, counting field marks, value marks, and subvalue marks until it finds the correct element to extract.)

See the REMOVE function for the function equivalent of this statement.

Examples

The first example sets the variable FIRST to the string MIKE and the variable X to 2 (field mark). The second example executes the REMOVE and PRINT statements until all the elements have been extracted, at which point A = 0. Printed lines are 12, 4, 5, 7654, and 00.

Source Lines	Program Output

`FM=CHAR(254)`
`NAME='MIKE':FM:'JOHN':FM`
`REMOVE FIRST FROM NAME SETTING X`
`PRINT 'X= ':X, 'FIRST= ':FIRST`
	`X= 2  FIRST= MIKE`

`VM=CHAR(253)`
`A=1`
`Z=12:VM:4:VM:5:VM:7654:VM:00`
`FOR X=1 TO 20 UNTIL A=0`
`  REMOVE Y FROM Z SETTING A`
`  PRINT 'Y= ':Y, 'A= ':A`
`NEXT X`
	`Y= 12  A= 3`
	`Y= 4  A= 3`
	`Y= 5  A= 3`
	`Y= 7654 A= 3`
	`Y= 0  A= 0`

REPEAT statement

The REPEAT statement is a loop-controlling statement. For syntax details, see the LOOP statement.

REPLACE function

Syntax

REPLACE (*expression*, *field#*, *value#*, *subvalue#* { , | ; } *replacement*)

REPLACE (*expression* [,*field#* [,*value#*]] ; *replacement*)

variable < *field#* [,*value#* [,*subvalue#*]] >

Description

Use the REPLACE function to return a copy of a dynamic array with the specified field, value, or subvalue replaced with new data.

expression specifies a dynamic array.

The expressions *field#*, *value#*, and *subvalue#* specify the type and position of the element to be replaced. These expressions are called delimiter expressions.

replacement specifies the value that the element is given.

The *value#* and *subvalue#* are optional. However, if either *subvalue#* or both *value#* and *subvalue#* are omitted, a semicolon (;) must precede *replacement*, as shown in the second syntax.

You can use angle brackets to replace data in dynamic arrays. Angle brackets to the left of an assignment operator change the specified data in the dynamic array according to the assignment operator. Angle brackets to the right of an assignment operator indicate that an EXTRACT function is to be performed (for examples, see the EXTRACT function in Volume 1).

variable specifies the dynamic array containing the data to be changed.

The three possible results of delimiter expressions are described as case 1, case 2, and case 3.

Case 1: Both *value#* and *subvalue#* are omitted or are specified as 0. A field is replaced by the value of *replacement*.

- If *field#* is positive and less than or equal to the number of fields in the dynamic array, the field specified by *field#* is replaced by the value of *replacement*.
- If *field#* is negative, a new field is created by appending a field mark and the value of *replacement* to the last field in the dynamic array.

- If *field#* is positive and greater than the number of fields in the dynamic array, a new field is created by appending the proper number of field marks, followed by the value of *replacement*; thus, the value of *field#* is the number of the new field.

Case 2: *subvalue#* is omitted or is specified as 0, and *value#* is nonzero. A value in the specified field is replaced with the value of *replacement*.

- If *value#* is positive and less than or equal to the number of values in the field, the value specified by the *value#* is replaced by the value of *replacement*.
- If *value#* is negative, a new value is created by appending a value mark and the value of *replacement* to the last value in the field.
- If *value#* is positive and greater than the number of values in the field, a value is created by appending the proper number of value marks, followed by the value of *replacement*, to the last value in the field; thus, the value of *value#* is the number of the new value in the specified field.

Case 3: *field#*, *value#*, and *subvalue#* are all specified and are nonzero. A subvalue in the specified value of the specified field is replaced with the value of *replacement*.

- If *subvalue#* is positive and less than or equal to the number of subvalues in the value, the subvalue specified by the *subvalue#* is replaced by the value of *replacement*.
- If *subvalue#* is negative, a new subvalue is created by appending a subvalue mark and the subvalue of *replacement* to the last subvalue in the value.
- If the *subvalue#* is positive and greater than the number of subvalues in the value, a new subvalue is created by appending the proper number of subvalue marks followed by the value of *replacement* to the last subvalue in the value; thus, the value of the expression *subvalue#* is the number of the new subvalue in the specified value.

In IDEAL, PICK, PIOPEN, and REALITY flavor accounts, if *replacement* is an empty string and an attempt is made to append the new element to the end of the dynamic array, field, or value, the dynamic array, field, or value is left unchanged; additional delimiters are not appended. Use the EXTRA.DELIM option of the

$OPTIONS statements to make the REPLACE function append a delimiter to the dynamic array, field, or value.

If *replacement* is the null value, the stored representation of null (CHAR(128)) is inserted into *dynamic.array*. If *dynamic.array* evaluates to the null value, it remains unchanged by the replacement. If the REPLACE statement references a subelement of an element whose value is the null value, the dynamic array is unchanged.

INFORMATION and IN2 Flavors

In INFORMATION and IN2 flavor accounts, if *expression* is an empty string and the new element is appended to the end of the dynamic array, the end of a field, or the end of a value, a delimiter is appended to the dynamic array, field, or value. Use the –EXTRA.DELIM option of the $OPTIONS statement to make the REPLACE function work as it does in IDEAL, PICK, and REALITY flavor accounts.

Examples

In the following examples a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

The first example replaces field 1 with # and sets Q to #FAVBVDSEFDFFF:

```
R=@FM:"A":@VM:"B":@VM:"D":@SM:"E":@FM:"D":@FM:@FM:"F"
Q=R
Q=REPLACE(Q,1;"#")
```

The next example replaces the first subvalue of the third value in field 2 with # and sets Q to FAVBV#SEFDFFF:

```
Q=R
Q<2,3,1>="#"
```

The next example replaces field 4 with # and sets Q to FAVBVDSEFDF#FF:

```
Q=R
Q=REPLACE(Q,4,0,0;"#")
```

The next example replaces the first value in fields 1 through 4 with # and sets Q to #F#VBVDSEF#F#FF:

```
Q=R
FOR X=1 TO 4
Q=REPLACE(Q,X,1,0;"#")
NEXT
```

The next example appends a value mark and # to the last value in field 2 and sets Q to FAvBvDsEv#FDFFF:

```
Q=R
Q=REPLACE(Q,2,-1;"#")
```

RETURN statement

Syntax

RETURN [TO *statement.label*]

Description

Use the RETURN statement to terminate a subroutine and return control to the calling program or statement.

If the TO clause is not specified, the RETURN statement exits either an internal subroutine called by a GOSUB statement or an external subroutine called by a CALL statement. Control returns to the statement that immediately follows the CALL or GOSUB statement.

Use a RETURN statement to terminate an internal subroutine called with a GOSUB statement to ensure that the program proceeds in the proper sequence.

Use a RETURN statement or an END statement to terminate an external subroutine called with a CALL statement. When you exit an external subroutine called by CALL, all files opened by the subroutine are closed, except files that are open to common variables.

Use the TO clause to exit only an internal subroutine; control passes to the specified statement label. If you use the TO clause and *statement.label* does not exist, an error message appears when the program is compiled.

Note: Using the TO clause can make program debugging and modification extremely difficult. Be careful when you use the RETURN TO statement, because all other GOSUBs or CALLs active at the time the GOSUB is executed remain active, and errors can result.

If the RETURN or RETURN TO statement does not have a place to return to, control is passed to the calling program or to the command language.

Example

In the following example, subroutine XYZ prints the message “THIS IS THE EXTERNAL SUBROUTINE” and returns to the main program:

```
20: GOSUB 80:
25: PRINT "THIS LINE WILL NOT PRINT"
30: PRINT "HI THERE"
40: CALL XYZ
```

```
60: PRINT "BACK IN MAIN PROGRAM"
70: STOP
80: PRINT "THIS IS THE INTERNAL SUBROUTINE"
90: RETURN TO 30:
END
```

This is the program output:

```
THIS IS THE INTERNAL SUBROUTINE
HI THERE
THIS IS THE EXTERNAL SUBROUTINE
BACK IN MAIN PROGRAM
```

RETURN (value) statement

Syntax

RETURN (*expression*)

Description

Use the RETURN (*value*) statement to return a value from a user-written function.

expression evaluates to the value you want the user-written function to return. If you use a RETURN (*value*) statement in a user-written function and you do not specify *expression*, an empty string is returned by default.

You can use the RETURN (*value*) statement only in user-written functions. If you use one in a program or subroutine, an error message appears.

Syntax

REUSE (*expression*)

Description

Use the REUSE function to specify that the value of the last field, value, or subvalue be reused in a dynamic array operation.

expression is either a dynamic array or an expression whose value is considered to be a dynamic array.

When a dynamic array operation processes two dynamic arrays in parallel, the operation is always performed on corresponding subvalues. This is true even for corresponding fields, each of which contains a single value. This single value is treated as the first and only subvalue in the first and only value in the field.

A dynamic array operation isolates the corresponding fields, values, and subvalues in a dynamic array. It then operates on them in the following order:

1. The subvalues in the values.
2. The values in the fields.
3. The fields of each dynamic array.

A dynamic array operation without the REUSE function adds zeros or empty strings to the shorter array until the two arrays are equal. (The DIVS function is an exception. If a divisor element is absent, the divisor array is padded with ones, so that the dividend value is returned.)

The REUSE function reuses the last value in the shorter array until all elements in the longer array are exhausted or until the next higher delimiter is encountered.

After all subvalues in a pair of corresponding values are processed, the dynamic array operation isolates the next pair of corresponding values in the corresponding fields and repeats the procedure.

After all values in a pair of corresponding fields are processed, the dynamic array operation isolates the next pair of corresponding fields in the dynamic arrays and repeats the procedure.

If *expression* evaluates to the null value, the null value is replicated, and null is returned for each corresponding element.

Example

```
B = (1:@SM:6:@VM:10:@SM:11)
A = ADDS(REUSE(5),B)
PRINT "REUSE(5) + 1:@SM:6:@VM:10:@SM:11 = ": A
*
PRINT "REUSE(1:@SM:2) + REUSE(10:@VM:20:@SM:30) = ":
PRINT ADDS(REUSE(1:@SM:2),REUSE(10:@VM:20:@SM:30))
*
PRINT "(4:@SM:7:@SM:8:@VM:10)*REUSE(10) = ":
PRINT MULS((4:@SM:7:@SM:8:@VM:10 ),REUSE(10))
```

This is the program output:

```
REUSE(5) + 1:@SM:6:@VM:10:@SM:11 = 6s11v15s16
REUSE(1:@SM:2) + REUSE(10:@VM:20:@SM:30) = 11s12v22s32
(4:@SM:7:@SM:8:@VM:10)*REUSE(10) = 40s70s80v100
```

Syntax

REVREMOVE *element* FROM *dynamic.array* SETTING *variable*

Description

Use the REVREMOVE statement to successively extract dynamic array elements that are separated by system delimiters. The elements are extracted from right to left, in the opposite order from those extracted by the REMOVE statement. When a system delimiter is encountered, the extracted element is assigned to *element*.

dynamic.array is an expression that evaluates to the dynamic array from which to extract elements.

variable is set to a code value corresponding to the system delimiter terminating the element just extracted. The delimiter code settings assigned to variable are as follows:

0 End of string

1 Item mark ASCII CHAR(255)

2 Field mark ASCII CHAR(254)

3 Value mark ASCII CHAR(253)

4 Subvalue mark ASCII CHAR(252)

5 Text mark ASCII CHAR(251)

6 ASCII CHAR(250)

7 ASCII CHAR(249)

8 ASCII CHAR(248)

The REVREMOVE statement extracts one element each time it is executed, beginning with the last element in *dynamic.array*. The operation can be repeated until all elements of *dynamic.array* are extracted. The REVREMOVE statement does not change the dynamic array.

As each element is extracted from *dynamic.array* to *element*, a pointer associated with *dynamic.array* moves back to the beginning of the element just extracted.

The pointer is reset to the beginning of *dynamic.array* whenever *dynamic.array* is reassigned. Therefore, *dynamic.array* should not be assigned a new value until all elements have been extracted (that is, until *variable* = 0).

If an element in *dynamic.array* is the null value, null is returned for that element.

Use REVREMOVE with the REMOVE statement. After a REMOVE statement, REVREMOVE returns the same string as the preceding REMOVE, setting the pointer to the delimiter preceding the extracted element. Thus, a subsequent REMOVE statement extracts the same element yet a third time.

If no REMOVE statement has been performed on *dynamic.array* or if the leftmost dynamic array element has been returned, *element* is set to the empty string and *variable* indicates end of string (that is, 0).

Example

```
DYN = "THIS":@FM:"HERE":@FM:"STRING"
REMOVE VAR FROM DYN SETTING X
PRINT VAR
REVREMOVE NVAR FROM DYN SETTING X
PRINT NVAR
REMOVE CVAR FROM DYN SETTING X
PRINT CVAR
```

The program output is:

```
THIS
THIS
THIS
```

Syntax

REWIND [UNIT (*mtu*)]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the REWIND statement to rewind a magnetic tape to the beginning-of-tape position.

The UNIT clause specifies the number of the tape drive unit. Tape unit 0 is used if no unit is specified. If the UNIT clause is used, *mtu* is an expression that evaluates to a code made up of three decimal digits. Although the *mtu* expression is a function of the UNIT clause, the REWIND statement uses only the third digit (the *u*). Its value must be in the range of 0 through 7. If *mtu* evaluates to the null value, the REWIND statement fails, the program terminates, and a run-time error message appears.

Before a REWIND statement is executed, a tape drive unit must be attached to the user. Use the ASSIGN command to assign a tape unit to a user. If no tape unit is attached or if the unit specification is incorrect, the ELSE statements are executed.

Use the STATUS function after a REWIND statement is executed, to determine the result of the operation. The value returned by the STATUS function for the REWIND statement is hardware-dependent (that is, it varies according to the characteristics of the specific tape drive). Consult the documentation that accompanied your tape drive for information on interpreting the values returned by the STATUS function.

PIOPEN Flavor

If you have a program that specifies the syntax UNIT *ndmtu*, the *nd* elements are ignored by the compiler and no errors are reported.

Example

```
REWIND UNIT(002) ELSE PRINT "UNIT NOT ATTACHED"
```

RIGHT function

Syntax

RIGHT (*string*, *n*)

Description

Use the RIGHT function to extract a substring comprising the last *n* characters of a string. It is equivalent to the following substring extraction operation:

string [*length*]

If you use this function, you need not calculate the string length.

If *string* evaluates to the null value, null is returned. If *n* evaluates to the null value, the RIGHT function fails, the program terminates, and a run-time error message appears.

Example

```
PRINT RIGHT("ABCDEFGH",3)
```

This is the program output:

```
FGH
```

Syntax

RND (*expression*)

Description

Use the RND function to generate any positive or negative random integer or 0.

expression evaluates to the total number of integers, including 0, from which the random number can be selected. That is, if *n* is the value of *expression*, the random number is generated from the numbers 0 through ($n - 1$).

If *expression* evaluates to a negative number, a random negative number is generated. If *expression* evaluates to 0, 0 is the random number. If *expression* evaluates to the null value, the RND function fails, the program terminates, and a run-time error message appears.

See the RANDOMIZE statement for details on generating repeatable sequences of random numbers.

Example

```
A=20
PRINT RND(A)
PRINT RND(A)
PRINT RND(A)
PRINT RND(A)
```

This is the program output:

```
10
3
6
10
```

ROLLBACK statement

Syntax

ROLLBACK [WORK] [THEN *statements*] [ELSE *statements*]

Description

Use the ROLLBACK statement to cancel all file I/O changes made during a transaction. The WORK keyword provides compatibility with SQL syntax conventions; it is ignored by the compiler.

A transaction includes all statements executed since the most recent BEGIN TRANSACTION statement. The ROLLBACK statement rolls back all changes made to files during the active transaction. If a subtransaction rolls back, none of the changes resulting from the active subtransaction affect the parent transaction. If the top-level transaction rolls back, none of the changes made are committed to disk, and the database remains unaffected by the transaction.

Use the ROLLBACK statement in a transaction without a COMMIT statement to review the results of a possible change. Doing so does not affect the parent transaction or the database. Executing a ROLLBACK statement ends the current transaction. After the transaction ends, execution continues with the statement following the next END TRANSACTION statement.

If no transaction is active, the ROLLBACK statement generates a run-time warning, and the ELSE statements are executed.

Example

This example begins a transaction that applies locks to rec1 and rec2. If errors occur (such as a failed READU or a failed WRITE), the ROLLBACK statements ensure that no changes are written to the file.

```
BEGIN TRANSACTION
   READU data1 FROM file1,rec1 ELSE ROLLBACK
   READU data2 FROM file2,rec2 ELSE ROLLBACK
       .
       .
       .
   WRITE new.data1 ON file1,rec1 ELSE ROLLBACK
   WRITE new.data2 ON file2,rec2 ELSE ROLLBACK
   COMMIT WORK
END TRANSACTION
```

Note that the update record lock on rec1 is not released on successful completion of the first WRITE statement.

RPC.CALL function

Syntax

RPC.CALL (*connection.ID*, *procedure*, *#args*, MAT *arg.list*, *#values*,
MAT *return.list*)

Description

Use the RPC.CALL function to make requests of a connected server. The request is packaged and sent to the server using the C client RPC library. RPC.CALL returns the results of processing the remote request: 1 for success, 0 for failure.

connection.ID is the handle of the open server connection on which to issue the RPC request. The RPC.CONNECT function gets the *connection.ID*.

procedure is a string identifying the operation requested of the server.

#args is the number of elements of *arg.list* to pass to the RPC server.

arg.list is a two-dimensional array (matrix) containing the input arguments to pass to the RPC server. The elements of this array represent ordered pairs of values. The first value is the number of the argument to the server operation, the second value is an argument-type declarator. (Data typing generalizes the RPC interface to work with servers that are data-type sensitive.)

#values is the number of values returned by the server.

return.list is a dimensioned array containing the results of the remote operation returned by RPC.CALL. Like *arg.list*, the results are ordered pairs of values.

RPC.CALL builds an RPC packet from *#args* and *arg.list*. Functions in the C client RPC library transmit the packet to the server and wait for the server to respond. When a response occurs, the RPC packet is separated into its elements and stored in the array *return.list*.

Use the STATUS function after an RPC.CALL function is executed to determine the result of the operation, as follows:

81001	Connection closed, reason unspecified.
81002	*connection.ID* does not correspond to a valid bound connection.
81004	Error occurred while trying to store an argument in the transmission packet.
81005	Procedure access denied because of a mismatch of RPC versions.
81008	Error occurred because of a bad parameter in *arg.list*.

81009 Unspecified RPC error.

81010 *#args* does not match expected argument count on remote machine.

81015 Timeout occurred while waiting for response from server.

Example

The following example looks for jobs owned by *fred*. The server connection was made using the RPC.CONNECT function.

```
args (1,1) = "fred"; args (1,2) = UVRPC.STRING
IF (RPC.CALL (server.handle, "COUNT.USERS", 1, MAT args,
       return.count, MAT res)) ELSE
   PRINT "COUNT.JOBS request failed, error code is: " STATUS()
   GOTO close.connection:
END
```

RPC.CONNECT function

Syntax

RPC.CONNECT (*host*, *server*)

Description

Use the RPC.CONNECT function to establish a connection to a server process. Once the host and server are identified, the local UV/Net daemon tries to connect to the remote server. If the attempt succeeds, RPC.CONNECT returns a connection ID. If it fails, RPC.CONNECT returns 0. The connection ID is a nonzero integer used to refer to the server in subsequent calls to RPC.CALL and RPC.DISCONNECT.

host is the name of the host where the server resides.

UNIX. This is defined in the local */etc/hosts* file.

Windows NT. This is defined in the *system32\drivers\etc\hosts* file.

server is the name, as defined in the remote */etc/services* file, of the RPC server class on the target host.

If *host* is not in the */etc/hosts* file, or if *service* is not in the remote */etc/services* file, the connection attempt fails.

Use the STATUS function after an RPC.CONNECT function is executed to determine the result of the operation, as follows:

81005	Connection failed because of a mismatch of RPC versions.
81007	Connection refused because the server cannot accept more clients.
81009	Unspecified RPC error.
81011	Host is not in the local */etc/hosts* file.
81012	Remote *uvrpcd* cannot start *service* because it could not fork the process.
81013	Cannot open the remote *uvrpcservices* file.
81014	Service not found in the remote *uvrpcservices* file.
81015	Connection attempt timed out.

Example

The following example connects to a remote server called MONITOR on HOST.A:

```
MAT args(1,2), res(1,2)
server.handle = RPC.CONNECT ("HOST.A", "MONITOR")
IF (server.handle = 0) THEN
   PRINT "Connection failed, error code is: ": STATUS()
   STOP
END
```

RPC.DISCONNECT function

Syntax

RPC.DISCONNECT (*connection.ID*)

Description

Use the RPC.DISCONNECT function to end an RPC session.

connection.ID is the RPC server connection you want to close.

RPC.DISCONNECT sends a request to end a connection to the server identified by *connection.ID*. When the server gets the request to disconnect, it performs any required termination processing. If the call is successful, RPC.DISCONNECT returns 1. If an error occurs, RPC.DISCONNECT returns 0.

Use the STATUS function after an RPC.DISCONNECT function is executed to determine the result of the operation, as follows:

81001	The connection was closed, reason unspecified.
81002	*connection.ID* does not correspond to a valid bound connection.
81009	Unspecified RPC error.

Example

The following example closes the connection to a remote server called MONITOR on HOST.A:

```
MAT args(1,2), res(1,2)
server.handle = RPC.CONNECT ("HOST.A", "MONITOR")
IF (server.handle = 0) THEN
   PRINT "Connection failed, error code is: ": STATUS()
   STOP
END
   .
   .
   .
close.connection:
IF (RPC.DISCONNECT (server.handle)) ELSE
   PRINT "Bizarre disconnect error, result code is: " STATUS()
END
```

Syntax

SADD (*string.number.1*, *string.number.2*)

Description

Use the SADD function to add two string numbers and return the result as a string number. You can use this function in any expression where a string or string number is valid, but not necessarily where a standard number is valid, because string numbers can exceed the range of numbers that standard arithmetic operators can handle.

Either string number can evaluate to any valid number or string number.

If either string number contains nonnumeric data, an error message is generated, and 0 replaces the nonnumeric data. If either string number evaluates to the null value, null is returned.

Example

```
A = 88888888888888888
B = 77777777777777777
X = "88888888888888888"
Y = "77777777777777777"
PRINT A + B
PRINT SADD(X,Y)
```

This is the program output:

```
166666666666667000
166666666666666665
```

SCMP function

Syntax

SCMP (*string.number.1*, *string.number.2*)

Description

Use the SCMP function to compare two string numbers and return one of the following three numbers: –1 (less than), 0 (equal), or 1 (greater than). If *string.number.1* is less than *string.number.2*, the result is –1. If they are equal, the result is 0. If *string.number.1* is greater than *string.number.2*, the result is 1. You can use this function in any expression where a string or string number is valid.

Either string number can be a valid number or string number. Computation is faster with string numbers.

If either string number contains nonnumeric data, an error message is generated and 0 is used instead of the nonnumeric data. If either string number evaluates to the empty string, null is returned.

Example

```
X = "123456789"
Y = "123456789"
IF SCMP(X,Y) = 0 THEN PRINT "X is equal to Y"
   ELSE PRINT "X is not equal to Y"
END
```

This is the program output:

```
X is equal to Y
```

Syntax

SDIV (*string.number.1*, *string.number.2* [,*precision*])

Description

Use the SDIV function to divide *string.number.1* by *string.number.2* and return the result as a string number. You can use this function in any expression where a string or string number is valid, but not necessarily where a standard number is valid, because string numbers can exceed the range of numbers which standard arithmetic operators can handle. Either string number can be a valid number or a string number.

precision specifies the number of places to the right of the decimal point. The default precision is 14.

If either string number contains nonnumeric data, an error message is generated and 0 is used for that number. If either string number evaluates to the null value, null is returned.

Example

```
X = "1"
Y = "3"
Z = SDIV (X,Y)
ZZ = SDIV (X,Y,20)
PRINT Z
PRINT ZZ
```

This is the program output:

```
0.33333333333333
0.33333333333333333333
```

SEEK statement

Syntax

SEEK *file.variable* [, *offset* [, *relto*]]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the SEEK statement to move the file pointer by an offset specified in bytes, relative to the current position, the beginning of the file, or the end of the file.

file.variable specifies a file previously opened for sequential access.

offset is the number of bytes before or after the reference position. A negative offset results in the pointer being moved before the position specified by *relto*. If *offset* is not specified, 0 is assumed.

Windows NT. Line endings in files are denoted by the character sequence RETURN + LINEFEED rather than the single LINEFEED used in UNIX files. The value of *offset* should take into account this extra byte on each line in Windows NT file systems.

The permissible values of *relto* and their meanings follow:

0 Relative to the beginning of the file

1 Relative to the current position

2 Relative to the end of the file

If *relto* is not specified, 0 is assumed.

If the pointer is moved, the THEN statements are executed and the ELSE statements are ignored. If the THEN statements are not specified, program execution continues with the next statement.

If the file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

If *file.variable*, *offset*, or *relto* evaluates to the null value, the SEEK statement fails, the program terminates, and a run-time error message appears.

Windows NT. If you use the OPENDEV statement to open a 1/4-inch cartridge tape (60 MB or 150 MB) for sequential processing, you can only move the file pointer to the beginning or the end of the data. For diskette drives, you can only move the file pointer to the start of the data.

Seeking beyond the end of the file and then writing creates a gap, or hole, in the file. This hole occupies no physical space, and reads from this part of the file return as ASCII CHAR 0 (neither the number nor the character 0).

For more information about sequential file processing, see the OPENSEQ, READSEQ, and WRITESEQ statements.

Example

The following example reads and prints the first line of RECORD4. Then the SEEK statement moves the pointer five bytes from the front of the file, then reads and prints the rest of the current line.

```
OPENSEQ 'FILE.E', 'RECORD4' TO FILE ELSE ABORT
READSEQ B FROM FILE THEN PRINT B
SEEK FILE,5, 0 THEN READSEQ A FROM FILE
THEN PRINT A ELSE ABORT
```

This is the program output:

```
FIRST LINE
 LINE
```

SEEK(ARG.) statement

Syntax

SEEK(ARG. [,*arg#*]) [THEN *statements*] [ELSE *statements*]

Description

Use the SEEK(ARG.) statement to move the command line argument pointer to the next command line argument from left to right, or to a command line argument specified by *arg#*. The command line is delimited by blanks, and the first argument is assumed to be the first word after the program name. When a cataloged program is invoked, the argument list starts with the second word in the command line.

Blanks in quoted strings are not treated as delimiters. A quoted string is treated as a single argument.

arg# specifies the command line argument to move to. It must evaluate to a number. If *arg#* is not specified, the pointer moves to the next command line argument. SEEK(ARG.) works similarly to GET(ARG.) except that SEEK(ARG.) makes no assignments.

THEN and ELSE statements are both optional. The THEN clause is executed if the argument is found. The ELSE clause is executed if the argument is not found. The SEEK(ARG.) statement fails if *arg#* evaluates to a number greater than the number of command line arguments or if the last argument has been assigned and a SEEK(ARG.) with no *arg#* is used. To move to the beginning of the argument list, set *arg#* to 1.

If *arg#* evaluates to the null value, the SEEK(ARG.) statement fails, the program terminates, and a run-time error message appears.

Example

If the command line is

```
RUN BP PROG ARG1 ARG2 ARG3
```

and the program is

```
A=5;B=2
SEEK(ARG.)
SEEK(ARG.,B)
SEEK(ARG.)
SEEK(ARG.,A-B)
SEEK(ARG.,1)
```

the system pointer moves as follows:

```
ARG2
ARG2
ARG3
ARG3
ARG1
```

SELECT statements

Syntax

SELECT [*variable*] [TO *list.number*] [ON ERROR *statements*]

SELECTN [*variable*] [TO *list.number*] [ON ERROR *statements*]

SELECTV [*variable*] TO *list.variable* [ON ERROR *statements*]

Description

Use a SELECT statement to create a numbered select list of record IDs from a UniVerse file or a dynamic array. A subsequent READNEXT statement can access this select list, removing one record ID at a time from the list. READNEXT instructions can begin processing the select list immediately.

variable can specify a dynamic array or a file variable. If it specifies a dynamic array, the record IDs must be separated by field marks (ASCII 254). If *variable* specifies a file variable, the file variable must have previously been opened. If *variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement). If the file is neither accessible nor open, or if *variable* evaluates to the null value, the SELECT statement fails, the program terminates and a runtime error message appears.

If the file is an SQL table, the effective user of the program must have SQL SELECT privilege to SELECT records in the file. See the AUTHORIZATION statement in Volume 1 for information about the effective user of a program.

You must use a file lock with the SELECT statement when it is within a transaction running at isolation level 4 (serializable). This prevents phantom reads.

The TO clause specifies the select list that is to be used. *list.number* is an integer from 0 through 10. If no *list.number* is specified, select list 0 is used.

The record IDs of all the records in the file, in their stored order, form the list. Each record ID is one entry in the list.

The SELECT statement does not process the entire file at once. It selects record IDs group by group. The @SELECTED variable is set to the number of elements in the group currently being processed.

You often want a select list with the record IDs in an order different from their stored order or with a subset of the record IDs selected by some specific criteria. To do this, use the SELECT or SSELECT commands in a BASIC EXECUTE statement. Processing the list by READNEXT is the same, regardless of how the list is created.

Use the SELECTV statement to store the select list in a named list variable instead of to a numbered select list. *list.variable* is an expression that evaluates to a valid variable name. This is the default behavior of the SELECT statement in PICK, REALITY, and IN2 flavor accounts. You can also use the VAR.SELECT option of the $OPTIONS statement to make the SELECT statement act as it does in PICK, REALITY, and IN2 flavor accounts.

The ON ERROR Clause

The ON ERROR clause is optional in the SELECT statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the SELECT statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

PICK, REALITY, and IN2 Flavors

In a PICK, REALITY, or IN2 flavor account, the SELECT statement has the following syntax:

SELECT[V] [*variable*] TO *list.variable*

SELECTN [*variable*] TO *list.number*

SELECT statements

You can use either the SELECT or the SELECTV statement to create a select list and store it in a named list variable. The only useful thing you can do with a list variable is use a READNEXT statement to read the next element of the select list.

Use the SELECTN statement to store the select list in a numbered select list. *list.number* is an expression that evaluates to a number from 0 through 10. You can also use the –VAR.SELECT option of the $OPTIONS statement to make the SELECT statement act as it does in IDEAL and INFORMATION flavor accounts.

Example

The following example opens the file SUN.MEMBER to the file variable MEMBER.F, then creates an active select list of record IDs. The READNEXT statement assigns the first record ID in the select list to the variable @ID, then prints it. Next, the file SUN.SPORT is opened to the file variable SPORT.F, and a select list of its record IDs is stored as select list 1. The READNEXT statement assigns the first record ID in the select list to the variable A, then prints DONE.

```
OPEN '','SUN.MEMBER' TO MEMBER.F ELSE PRINT "NOT OPEN"
SELECT
READNEXT @ID THEN PRINT @ID
*
OPEN '','SUN.SPORT' TO SPORT.F ELSE PRINT "NOT OPEN"
SELECT TO 1
READNEXT A FROM 1 THEN PRINT "DONE" ELSE PRINT "NOT"
```

This is the program output:

```
4108
DONE
```

Syntax

SELECTE TO *list.variable*

Description

Use the SELECTE statement to assign the contents of select list 0 to *list.variable*. *list.variable* is activated in place of select list 0 and can be read with the READNEXT statement.

SELECTINDEX statement

Syntax

SELECTINDEX *index* [,*alt.key*] FROM *file.variable* [TO *list.number*]

Description

Use the SELECTINDEX statement to create select lists from secondary indices.

file.variable specifies an open file.

index is an expression that evaluates to the name of an indexed field in the file. *index* must be the name of the field that was used in the CREATE.INDEX command when the index was built.

list.number is an expression that evaluates to the select list number. It can be a number from 0 through 10. The default list number is 0.

alt.key is an expression that evaluates to an alternate key value contained in the index. If *alt.key* is specified, a select list is created of the record IDs referenced by *alt.key*. If *alt.key* is not specified, a select list is created of all of the index's alternate keys.

If the field is not indexed, the select list is empty, and the value of the STATUS function is 1; otherwise the STATUS function is 0. If *index*, *alt.key*, or *file.variable* evaluates to the null value, the SELECTINDEX statement fails, the program terminates, and a run-time error message appears.

PIOPEN Flavor

In a PIOPEN flavor account, the SELECTINDEX statement creates a select list from the secondary indices without duplicate keys. To implement this functionality in other flavors, use the PIOPEN.SELIDX option with the $OPTION statement.

Example

In the following example, the first SELECTINDEX selects all data values to list 1. The second SELECTINDEX selects record IDs referenced by STOREDVAL to list 2.

```
OPEN "", "DB" TO FV ELSE STOP "OPEN FAILED"
SELECTINDEX "F1" FROM FV TO 1
EOV = 0
LOOP
   SELECTINDEX "F1" FROM FV TO 1
```

```
UNTIL EOV DO
   SELECTINDEX "F1", STOREDVAL FROM FV TO 2
   EOK = 0
   LOOP
      READNEXT KEY FROM 2 ELSE EOK=1
   UNTIL EOK DO
      PRINT "KEY IS ":KEY:" STOREDVAL IS ":STOREDVAL
   REPEAT
REPEAT
END
```

SELECTINFO function

Syntax

SELECTINFO (*list*, *key*)

Description

Use the SELECTINFO function to determine whether a select list is active, or to determine the number of items it contains.

list is an expression evaluating to the number of the select list for which you require information. The select list number must be in the range of 0 through 10.

key specifies the type of information you require. You can use equate names for the keys as follows:

IK$SLACTIVE	Returns 1 if the select list specified is active, and returns 0 if the select list specified is not active.
IK$SLCOUNT	Returns the number of items in the select list. 0 is returned if the select list is not active or is an empty select list.

Equate Names

An insert file of equate names is provided for the SELECTINFO keys. To use the equate names, specify the directive $INCLUDE SYSCOM INFO_KEYS.INS.IBAS when you compile your program.

Example

In the following example, the insert file containing the equate name is inserted by the $INCLUDE statement. The conditional statement tests if select list 0 is active.

```
$INCLUDE SYSCOM INFO_KEYS.INS.IBAS
   IF SELECTINFO(0,IK$SLACTIVE)
     THEN PRINT 'SELECT LIST ACTIVE'
     ELSE PRINT 'SELECT LIST NOT ACTIVE'
   END
```

Syntax

SEND *output.expr* [:] TO *device*
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the SEND statement to write a block of data to a device. The SEND statement can be used to write data to a device that has been opened for I/O using the OPENDEV or OPENSEQ statement.

output.expr is an expression evaluating to a data string that will be written to *device*. If the optional colon is used after *output.expr*, the terminating newline is not generated.

device is a valid file variable resulting from a successful OPENDEV or OPENSEQ statement. This is the handle to the I/O device that supplies the data stream for the operation of the SEND statement.

The SEND syntax requires that either a THEN or an ELSE clause, or both, be specified. If data is successfully sent, the SEND statement executes the THEN clause. If data cannot be sent, it executes the ELSE clause.

The data block specified by *output.expr* is written to the device followed by a newline. Upon successful completion of the SEND operation, program control is passed to the THEN clause if specified. If an error occurs during the SEND operation, program control is passed to the ELSE clause if specified.

Example

The following code fragment shows how the SEND statement is used to write a series of messages on a connected device:

```
OPENDEV "TTY10" TO TTYLINE ELSE STOP "CANNOT OPEN TTY10"
LOOP
       INPUT MESSAGE
   WHILE MESSAGE # "QUIT" DO
       SEND MESSAGE TO TTYLINE
       ELSE
          STOP "ERROR WRITING DATA TO TTY10"
       END
REPEAT
```

SENTENCE function

Syntax

SENTENCE ( )

Description

Use the SENTENCE function to return the stored sentence that invoked the current process. Although the SENTENCE function uses no arguments, parentheses are required to identify it as a function. The SENTENCE function is a synonym for the @SENTENCE system variable.

A PERFORM statement in a program updates the system variable, @SENTENCE, with the command specified in the PERFORM statement.

Example

```
PRINT SENTENCE()
```

This is the program output:

```
RUN BP TESTPROGRAM
```

Syntax

SEQ (*expression*)

Description

Use the SEQ function to convert an ASCII character to its numeric string equivalent.

expression evaluates to the ASCII character to be converted. If *expression* evaluates to the null value, null is returned.

The SEQ function is the inverse of the CHAR function.

PICK, IN2, and REALITY Flavors

In PICK, IN2, and REALITY flavors SEQ(" ") is 255 instead of 0. In IDEAL and INFORMATION flavor accounts, use the SEQ.255 option of the $OPTIONS statement to cause SEQ(" ") to be interpreted as 255.

Example

```
G="T"
A=SEQ(G)
PRINT A, A+1
PRINT SEQ("G")
```

This is the program output:

```
84    85
71
```

SEQS function

Syntax

SEQS (*dynamic.array*)

CALL –SEQS (*return.array, dynamic.array*)

CALL !SEQS (*return.array, dynamic.array*)

Description

Use the SEQS function to convert a dynamic array of ASCII characters to their numeric string equivalents.

dynamic.array specifies the ASCII characters to be converted. If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

Example

```
G="T":@VM:"G"
A=SEQS(G)
PRINT A
PRINT SEQS("G")
```

This is the program output:

```
84v71
71
```

SET TRANSACTION ISOLATION LEVEL statement

Syntax

SET TRANSACTION ISOLATION LEVEL *level*

Description

Use the SET TRANSACTION ISOLATION LEVEL statement to set the default transaction isolation level you need for your program.

Note: The isolation level you set with this statement remains in effect until another such statement is issued. This affects all activities in the session, including UniVerse commands and SQL transactions.

The SET TRANSACTION ISOLATION LEVEL statement cannot be executed while a transaction exists. Attempting to do so results in a run-time error message, program failure, and the rolling back of all uncommitted transactions started in the execution environment.

level has the following syntax:

{*n* | *keyword* | *expression*}

level is an expression that evaluates to 0 through 4, or one of the following keywords:

Effects of ISOLATION LEVELs on Transactions

Integer	Keyword	Effect on This Transaction
0	NO.ISOLATION	Prevents lost updates.[1]
1	READ.UNCOMMITTED	Prevents lost updates.
2	READ.COMMITTED	Prevents lost updates and dirty reads.
3	REPEATABLE.READ	Prevents lost updates, dirty reads, and nonrepeatable reads.
4	SERIALIZABLE	Prevents lost updates, dirty reads, nonrepeatable reads, and phantom writes.

1. Lost updates are prevented if the ISOMODE configurable parameter is set to 1 or 2.

Examples

The following example sets the default isolation level to 3 then starts a transaction at isolation level 4. The isolation level is reset to 3 after the transaction finishes.

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE.READ
PRINT "We are at isolation level 3."
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE
          PRINT "We are at isolation level 4."
          COMMIT WORK
END TRANSACTION
PRINT "We are at isolation level 3"
```

The next example uses an expression to set the transaction level:

```
PRINT "Enter desired transaction isolation level:":
INPUT TL
   SET TRANSACTION LEVEL TL
   BEGIN TRANSACTION
      {statements}
   END TRANSACTION
```

Syntax

SETREM *position* ON *dynamic.array*

Description

Use the SETREM statement to set the remove pointer in *dynamic.array* to the position specified by *position*.

position is an expression that evaluates to an integer. If it is larger than the length of *dynamic.array*, the length of *dynamic.array* is used. If it is less than 0, 0 is used.

dynamic.array must be a variable that evaluates to a string. If it does not evaluate to a string, an improper data type warning is issued.

If the pointer does not point to the first character after a system delimiter, subsequent REMOVE and REVREMOVE statements act as follows:

- A REMOVE statement returns a substring, starting from the pointer and ending at the next delimiter.
- A REVREMOVE statement returns a substring, starting from the previous delimiter and ending at the pointer.

Example

```
DYN = "THIS":@FM:"HERE":@FM:"STRING"
REMOVE VAR FROM DYN SETTING X
A = GETREM(DYN)
REMOVE VAR FROM DYN SETTING X
PRINT VAR
SETREM A ON DYN
REMOVE VAR FROM DYN SETTING X
PRINT VAR
```

The program output is:

```
HERE
HERE
```

SIN function

Syntax

SIN (*expression*)

Description

Use the SIN function to return the trigonometric sine of an expression. *expression* represents the angle expressed in degrees. Numbers greater than 1E17 produce a warning message, and 0 is returned. If *expression* evaluates to the null value, null is returned.

Example

```
PRINT SIN(45)
```

This is the program output:

```
0.7071
```

Syntax

SINH (*expression*)

Description

Use the SINH function to return the hyperbolic sine of *expression*. *expression* must be numeric and represents the angle expressed in degrees. If *expression* evaluates to the null value, null is returned.

Example

```
PRINT "SINH(2) = ":SINH(2)
```

This is the program output:

```
SINH(2) = 3.6269
```

SLEEP statement

Syntax

SLEEP [*seconds*]

Description

Use the SLEEP statement to suspend execution of a BASIC program, pausing for a specified number of seconds.

seconds is an expression evaluating to the number of seconds for the pause. If *seconds* is not specified, a value of 1 is used. If *seconds* evaluates to the null value, it is ignored and 1 is used.

Example

In the following example the program pauses for three seconds before executing the statement after the SLEEP statement. The EXECUTE statement clears the screen.

```
PRINT "STUDY THE FOLLOWING SENTENCE CLOSELY:"
PRINT
PRINT
PRINT "There are many books in the"
PRINT "the library."
SLEEP 3
EXECUTE 'CS'
PRINT "DID YOU SEE THE MISTAKE?"
```

This is the program output:

```
STUDY THE FOLLOWING SENTENCE CLOSELY:


There are many books in the
the library.
DID YOU SEE THE MISTAKE?
```

Syntax

SMUL (*string.number.1*, *string.number.2*)

Description

Use the SMUL function to multiply two string numbers and return the result as a string number. You can use this function in any expression where a string or string number is valid, but not necessarily where a standard number is valid, because string numbers can exceed the range of numbers that standard arithmetic operators can handle.

Either string number can be any valid number or string number.

If either string number contains nonnumeric data, an error message is generated and 0 is used for that number. If either string number evaluates to the null value, null is returned.

Example

```
X = "5436"
Y = "234"
Z = SMUL (X,Y)
PRINT Z
```

This is the program output:

```
1272024
```

SOUNDEX function

Syntax

SOUNDEX (*expression*)

Description

The SOUNDEX function evaluates *expression* and returns the most significant letter in the input string followed by a phonetic code. Nonalphabetic characters are ignored. If *expression* evaluates to the null value, null is returned.

This function uses the soundex algorithm (the same as the one used by the SAID keyword in RetrieVe) to analyze the input string. The soundex algorithm returns the first letter of the alphabetic string followed by a one- to three-digit phonetic code.

Example

Source Lines	Program Output

`DATA "MCDONALD", "MACDONALD", "MACDOUGALL"`
`FOR I=1 TO 3`
`  INPUT CUSTOMER`
`  PHONETIC.CODE=SOUNDEX(CUSTOMER)`
`  PRINT PHONETIC.CODE`
`NEXT`
	`?MCDONALD`
	`M235`
	`?MACDONALD`
	`M235`
	`?MACDOUGALL`
	`M232`

Syntax

SPACE (*expression*)

Description

Use the SPACE function to return a string composed of blank spaces. *expression* specifies the number of spaces in the string. If *expression* evaluates to the null value, the SPACE function fails, the program terminates, and a run-time error message appears.

There is no limit to the number of blank spaces that can be generated.

Example

```
PRINT "HI":SPACE(20):"THERE"
*
*
VAR=SPACE(5)
PRINT "TODAY IS":VAR:OCONV(DATE(),"D")
```

This is the program output:

```
HI          THERE
TODAY IS   18 JUN 1992
```

SPACES function

Syntax

SPACES (*dynamic.array*)

CALL –SPACES (*return.array, dynamic.array*)

CALL !SPACES (*return.array, dynamic.array*)

Description

Use the SPACES function to return a dynamic array with elements composed of blank spaces. *dynamic.array* specifies the number of spaces in each element. If *dynamic.array* or any element of *dynamic.array* evaluates to the null value, the SPACES function fails, the program terminates, and a run-time error message appears.

There is no limit to the number of blank spaces that can be generated except available memory.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

Syntax

SPLICE (*array1*, *expression*, *array2*)

CALL –SPLICE (*return.array*, *array1*, *expression*, *array2*)

CALL !SPLICE (*return.array*, *array1*, *expression*, *array2*)

Description

Use the SPLICE function to create a dynamic array of the element-by-element concatenation of two dynamic arrays, separating concatenated elements by the value of *expression*.

Each element of *array1* is concatenated with *expression* and with the corresponding element of *array2*. The result is returned in the corresponding element of a new dynamic array. If an element of one dynamic array has no corresponding element in the other dynamic array, the element is returned properly concatenated with *expression*. If either element of a corresponding pair is the null value, null is returned for that element. If *expression* evaluates to the null value, null is returned for the entire dynamic array.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A="A":@VM:"B":@SM:"C"
B="D":@SM:"E":@VM:"F"
C='-'
PRINT SPLICE(A,C,B)
```

This is the program output:

```
A-Ds-EvB-FsC-
```

SQRT function

Syntax

SQRT (*expression*)

Description

Use the SQRT function to return the square root of *expression*. The *expression* must evaluate to a numeric value that is greater than or equal to 0. If *expression* evaluates to a negative value, the result of the function is SQRT(–*n*) and an error message is printed. If *expression* evaluates to the null value, null is returned.

Example

```
A=SQRT(144)
PRINT A
*
PRINT "SQRT(45) IS ":SQRT(45)
```

This is the program output:

```
12
SQRT(45) IS 6.7082
```

Syntax

SQUOTE (*expression*)

CALL !SQUOTE (*quoted.expression*, *expression*)

Description

Use the SQUOTE function to enclose an expression in single quotation marks. If *expression* evaluates to the null value, null is returned, without quotation marks.

quoted.expression is the quoted string.

expression is the input string.

Example

```
PRINT SQUOTE(12 + 5) : " IS THE ANSWER."
END
```

This is the program output:

```
'17' IS THE ANSWER.
```

SSELECT statements

Syntax

SSELECT [*variable*] [TO *list.number*] [ON ERROR *statements*]

SSELECTN [*variable*] [TO *list.number*] [ON ERROR *statements*]

SSELECTV [*variable*] TO *list.variable* [ON ERROR *statements*]

Description

Use an SSELECT statement to create a numbered select list of record IDs in sorted order from a UniVerse file or a dynamic array. You can then access this select list by a subsequent READNEXT statement which removes one record ID at a time from the list.

variable can specify a dynamic array or a file variable. If it specifies a dynamic array, the record IDs must be separated by field marks (ASCII 254). If *variable* specifies a file variable, the file variable must have previously been opened. If *variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement). If the file is neither accessible nor open, or if *variable* evaluates to the null value, the SSELECT statement fails, the program terminates and a run-time error message appears.

If the file is an SQL table, the effective user of the program must have SQL SELECT privilege to SSELECT records in the file. See the AUTHORIZATION statement in Volume 1 for information about the effective user of a program.

You must use a file lock with the SSELECT statement when it is within a transaction running at isolation level 4 (serializable). This prevents phantom reads.

The TO clause specifies the select list that is to be used. *list.number* is an integer from 0 through 10. If no *list.number* is specified, select list 0 is used.

The record IDs of all the records in the file form the list. The record IDs are listed in ascending order. Each record ID is one entry in the list.

You often want a select list with the record IDs in an order different from their stored order or with a subset of the record IDs selected by some specific criteria. To do this, use the SELECT or SSELECT commands in a BASIC EXECUTE statement. Processing the list by READNEXT is the same, regardless of how the list is created.

Use the SSELECTV statement to store the select list in a named list variable instead of to a numbered select list. *list.variable* is an expression that evaluates to a valid variable name. This is the default behavior of the SSELECT statement in PICK,

REALITY, and IN2 flavor accounts. You can also use the VAR.SELECT option of the $OPTIONS statement to make the SSELECT statement act as it does in PICK, REALITY, and IN2 flavor accounts.

The ON ERROR Clause

The ON ERROR clause is optional in SSELECT statements. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of a SSELECT statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

PICK, REALITY, and IN2 Flavors

In a PICK, REALITY, or IN2 flavor account, the SSELECT statement has the following syntax:

SSELECT[V] [*variable*] TO *list.variable*

SSELECTN [*variable*] TO *list.number*

You can use either the SSELECT or the SSELECTV statement to create a select list and store it in a named list variable. The only useful thing you can do with a list variable is use a READNEXT statement to read the next element of the select list.

Use the SSELECTN statement to store the select list in a numbered select list. *list.number* is an expression that evaluates to a number from 0 through 10. You can also use the –VAR.SELECT option of the $OPTIONS statement to make the SSELECT statement act as it does in IDEAL and INFORMATION flavor accounts.

Example

The following example opens the file SUN.MEMBER to the file variable MEMBER.F, then creates an active sorted select list of record IDs. The READNEXT statement assigns the first record ID in the select list to the variable @ID, then prints it. Next, the file SUN.SPORT is opened to the file variable SPORT.F, and a sorted select list of its record IDs is stored as select list 1. The READNEXT statement assigns the first record ID in the select list to the variable A, then prints DONE.

```
OPEN '','SUN.MEMBER' ELSE PRINT "NOT OPEN"
SSELECT
READNEXT @ID THEN PRINT @ID
*
OPEN '','SUN.SPORT' ELSE PRINT "NOT OPEN"
SSELECT TO 1
READNEXT A FROM 1 THEN PRINT "DONE" ELSE PRINT "NOT"
```

This is the program output:

```
0001
DONE
```

Syntax

SSUB (*string.number.1*, *string.number.2*)

Description

Use the SSUB function to subtract *string.number.2* from *string.number.1* and return the result as a string number. You can use this function in any expression where a string or string number is valid, but not necessarily where a standard number is valid, because string numbers can exceed the range of numbers that standard arithmetic operators can handle.

Either string number can be any valid number or string number.

If either string number contains nonnumeric data, an error message is generated, and 0 replaces the nonnumeric data. If either string number evaluates to the null value, null is returned.

Example

```
X = "123456"
Y = "225"
Z = SSUB (X,Y)
PRINT Z
```

This is the program output:

```
123231
```

STATUS function

Syntax

STATUS ( )

Description

Use the STATUS function to determine the results of the operations performed by certain statements and functions.

The parentheses must be used with the STATUS function to distinguish it from potential user-named variables called STATUS. However, no arguments are required with the STATUS function.

The following sections describe STATUS function values.

After a BSCAN statement:

0 The scan proceeded beyond the leftmost or rightmost leaf node. *ID.variable* and *rec.variable* are set to empty strings.

1 The scan returned an existing record ID, or a record ID that matches *record*.

2 The scan returned a record ID that does not match *record*. *ID.variable* is either the next or the previous record ID in the B-tree, depending on the direction of the scan.

3 The file is not a B-tree (type 25) file, or, if the USING clause is used, the file has no active secondary indices.

4 *indexname* does not exist.

5 *seq* does not evaluate to A or D.

6 The index specified by *indexname* needs to be built.

10 An internal error was detected.

After a DELETE statement: After a DELETE statement with an ON ERROR clause, the value returned is the error number.

After a FILELOCK statement: After a FILELOCK statement with a LOCKED clause, the value returned is the terminal number of the user who has a conflicting lock.

After an FMT function:

0 The conversion is successful.

1 The string expression passed as an argument is invalid.

2 The conversion code passed as an argument to the function is invalid.

After a GET or GETX statement:

0	The timeout limit expired.
Any nonzero value	A device input error occurred.

After an ICONV or OCONV function:

0 The conversion is successful.

1 The string expression passed as an argument to the function is not convertible using the conversion code passed. An empty string is returned as the value of the function.

2 The conversion code passed as an argument to the function is invalid. An empty string is returned as the value of the function.

3 Successful conversion of a possibly invalid date.

After an INPUT @ statement: A 0 is returned if the statement was completed by a Return. The trap number is returned if the statement was completed by one of the trapped keys (see the INPUT @ and KEYTRAP statements in Volume 1).

After a MATWRITE, WRITE, WRITEU, WRITEV, or WRITEVU statement:

0 The record was locked before the operation.

–2 The record was unlocked before the operation.

–3 The record failed an SQL integrity check.

After an OPEN, OPENCHECK, OPENPATH, or OPENSEQ statement: The file type is returned if the file is opened successfully. If the file is not opened successfully, the following values may return:

Value	Description
–1	The filename was not found in the VOC file.
–2[1]	The filename or file is null. This error may also occur when you cannot open a file across UV/Net.

Value	Description
–3	An operating system access error occurs when you do not have permission to access a UniVerse file in a directory. For example, this error may occur when trying to access a type 1 or type 30 file.
–4[1]	An access error appears when you do not have operating system permissions or if DATA.30 is missing for a type 30 file.
–5	The operating system detected a read error.
–6	The lock file header cannot be unlocked.
–7	Invalid file revision or wrong byte-ordering exists for the platform.
–8[1]	Invalid part file information exists.
–9[1]	Invalid type 30 file information exists in a distributed file.
–10	A problem occurred while the file was being rolled forward during warmstart recovery. Therefore, the file is marked “inconsistent.”
–11	The file is a view; therefore it cannot be opened by a BASIC program.
–12	No SQL privileges exist to open the table.
–13[1]	An index problem exists.
–14	The NFS file cannot be opened.

1. A generic error that can occur for various reasons.

After a READ statement: If the file is a distributed file, the STATUS function returns the following:

–1 The partitioning algorithm does not evaluate to an integer.

–2 The part number is invalid.

After a READBLK statement:

0 The read is successful.

1 The end of file is encountered, or the number of bytes passed in was less than or equal to 0.

2 A timeout ended the read.

–1 The file is not open for a read.

After a READL, READU, READVL, or READVU statement: If the statement includes the LOCKED clause, the returned value is the terminal number, as returned by the WHO command, of the user who set the lock.

After a READSEQ statement:

0 The read is successful.

1 The end of file is encountered, or the number of bytes passed in was less than or equal to 0.

2 A timeout ended the read.

–1 The file is not open for a read.

After a READT, REWIND, WEOF, or WRITET statement: The returned value is hardware-dependent (that is, it varies according to the characteristics of the specific tape drive unit). Consult the documentation that accompanied your tape drive unit for information about interpreting the values returned by the STATUS function.

After an RPC.CALL, RPC.CONNECT, or RPC.DISCONNECT statement:

81001 A connection was closed for an unspecified reason.

81002 *connection.ID* does not correspond to a valid bound connection.

81004 Error occurred while trying to store an argument in the transmission packet.

81005 Procedure access denied because of a mismatch of RPC versions.

81007 Connection refused because the server cannot accept more clients.

81008 Error occurred because of a bad parameter in *arg.list*.

81009 An unspecified RPC error occurred.

81010 *#args* does not match the expected argument count on the remote machine.

81011 Host was not found in local */etc/hosts* file.

81012 Remote *uvrpcd* cannot start the service because it could not fork the process.

81013 The remote *uvrpcservices* file cannot be opened.

81014 Service was not found in the remote *uvrpcservices* file.

81015 A timeout occurred while waiting for a response from the server.

Example

Source Lines	Program Output

`OPEN '','EX.BASIC' TO FILE ELSE STOP`
`PRINT 'STATUS() IS ':STATUS()`
 	
`Q=123456`
`Q=OCONV(Q,"MD2")`
`PRINT 'STATUS() IS ':STATUS()`
 	
`Q='ASDF'`
`Q=OCONV(Q,"D2/")`
`PRINT 'STATUS() IS ':STATUS()`
	`STATUS() IS 1`
	`STATUS() IS 0`
	`STATUS() IS 1`

Syntax

STATUS *dynamic.array* FROM *file.variable*
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the STATUS statement to determine the status of an open file. The STATUS statement returns the file status as a dynamic array and assigns it to *dynamic.array.*

The following table lists the values of the dynamic array returned by the STATUS statement:

STATUS Statement Values

Field	Stored Value	Description
1	Current position in the file	Offset in bytes from beginning of file.
2	End of file reached	1 if EOF, 0 if not.
3	Error accessing file	1 if error, 0 if not.
4	Number of bytes available to read	
5	File mode	Permissions (convert to octal). **Windows NT.** This is the UNIX owner-group-other format as converted from the full Windows NT ACL format by the C runtime libraries.
6	File size	In bytes.
7	Number of hard links	0 if no links. **Windows NT.** The value is always 1 on non-NTFS partitions, > 0 on NTFS partitions.
8	User ID of owner	**UNIX.** The number assigned in */etc/passwd.* **Windows NT.** It is a UniVerse pseudo user ID based on the user name and domain of the user.
9	Group ID of owner	**UNIX.** The number assigned in */etc/passwd* **Windows NT.** It is always 0.

STATUS Statement Values (Continued)

Field	Stored Value	Description
10	I-node number	Unique ID of file on file system; on Windows NT the value is the Pelican internal version of the i-node for a file.
11	Device on which i-node resides	Number of device. The value is an internally calculated value on Windows NT.
12	Device for special character or block	Number of device. The value is the drive number of the disk containing the file on Windows NT.
13	Time of last access	Time in internal format.
14	Date of last access	Date in internal format.
15	Time of last modification	Time in internal format.
16	Date of last modification	Date in internal format.
17	Time and date of last status change	Time and date in internal format. On Windows NT it is the time the file was created.
18	Date of last status change	Date in internal format. On Windows NT it is the date the file was created.
19	Number of bytes left in output queue (applicable to terminals only)	
20	Operating system filename	The internal pathname UniVerse uses to access the file.
21	UniVerse file type	For file types 1–19, 25, or 30.
22	UniVerse file modulo	For file types 2–18 only.
23	UniVerse file separation	For file types 2–18 only.
24	Part numbers of part files belonging to a distributed file	Multivalued list. If file is a part file, this field contains the part number, and field 25 is empty.
25	Pathnames of part files belonging to a distributed file	Multivalued list. If file is a part file, this field is empty.

STATUS Statement Values (Continued)

Field	Stored Value	Description
26	Filenames of part files belonging to a distributed file	Multivalued list. If file is a part file, this field is empty.
27	Full pathname	The full pathname of the file. On Windows NT, the value begins with the UNC share name, if available; if not, the drive letter.
28	Integer from 1 through 7	SQL file privileges: 1 write-only 2 read-only 3 read / write 4 delete-only 5 delete / write 6 delete / read 7 delete / read / write
29		1 if no SQL file privileges, 0 if not.
30	User name	User name of the owner of the file.

file.variable specifies an open file. If *file.variable* evaluates to the null value, the STATUS statement fails, the program terminates, and a run-time error message appears.

If the STATUS array is assigned to *dynamic.array*, the THEN statements are executed and the ELSE statements are ignored. If no THEN statements are present, program execution continues with the next statement. If the attempt to assign the array fails, the ELSE statements are executed; any THEN statements are ignored.

Example

```
OPENSEQ '/etc/passwd' TO test THEN PRINT "File Opened" ELSE
ABORT
STATUS stat FROM test THEN PRINT stat
field5 = stat<5,1,1>
field6 = stat<6,1,1>
field8 = stat<8,1,1>
PRINT "permissions:": field5
PRINT "filesize:": field6
```

```
PRINT "userid:": field8
CLOSESEQ test
```

This is the program output:

```
File Opened
0F0F0F4164F33188F4164F1F0F2F2303F
   0F6856F59264F6590F42496F6588F42496F6588
   F0F/etc/passwdF0F0F0
permissions:33188
filesize:4164
userid:0
```

Syntax

STOP [*expression*]

STOPE [*expression*]

STOPM [*expression*]

Description

Use the STOP statement to terminate program execution and return system control to the calling environment, which can be a menu, a paragraph, another BASIC program, or the UniVerse command processor.

When *expression* is specified, its value is displayed before the STOP statement is executed. If *expression* evaluates to the null value, nothing is printed.

To stop all processes and return to the command level, use the ABORT statement.

Use the ERRMSG statement if you want to display a formatted error message from the ERRMSG file when the program stops.

STOPE and STOPM Statements

The STOPE statement uses the ERRMSG file for error messages instead of using text specified by *expression*. The STOPM statement uses text specified by *expression* rather than messages in the ERRMSG file. If *expression* in the STOPE statement evaluates to the null value, the default error message is printed:

```
Message ID is NULL: undefined error
```

PICK, IN2, and REALITY Flavors

In PICK, IN2, and REALITY flavor accounts, the STOP statement uses the ERRMSG file for error messages instead of using text specified by *expression*. Use the STOP.MSG option of the $OPTIONS statement to get this behavior in IDEAL and INFORMATION flavor accounts.

Example

```
PRINT "1+2=":1+2
STOP "THIS IS THE END"
```

This is the program output:

```
1+2=3
THIS IS THE END
```

STORAGE statement

Syntax

STORAGE *arg1 arg2 arg3*

Description

The STORAGE statement performs no function. It is provided for compatibility with other Pick systems.

STR function

Syntax

STR (*string*, *repeat*)

Description

Use the STR function to produce a specified number of repetitions of a particular character string.

string is an expression that evaluates to the string to be generated.

repeat is an expression that evaluates to the number of times *string* is to be repeated. If *repeat* does not evaluate to a value that can be truncated to a positive integer, an empty string is returned.

If *string* evaluates to the null value, null is returned. If *repeat* evaluates to the null value, the STR function fails, the program terminates, and a run-time error message appears.

Example

```
PRINT STR('A',10)
*
X=STR(5,2)
PRINT X
*
X="HA"
PRINT STR(X,7)
```

This is the program output:

```
AAAAAAAAAA
55
HAHAHAHAHAHAHA
```

Syntax

STRS (*dynamic.array, repeat*)

CALL –STRS (*return.array, dynamic.array, repeat*)

CALL !STRS (*return.array, dynamic.array, repeat*)

Description

Use the STRS function to produce a dynamic array containing the specified number of repetitions of each element of *dynamic.array.*

dynamic.array is an expression that evaluates to the strings to be generated.

repeat is an expression that evaluates to the number of times the elements are to be repeated. If it does not evaluate to a value that can be truncated to a positive integer, an empty string is returned for *dynamic.array.*

If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is the null value, null is returned for that element. If *repeat* evaluates to the null value, the STRS function fails, the program terminates, and a run-time error message appears.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

Example

```
ABC="A":@VM:"B":@VM:"C"
PRINT STRS(ABC,3)
```

This is the program output:

```
AAAvBBBvCCC
```

SUBR function

Syntax

SUBR (*name*, [*argument* [,*argument* ...]])

Description

Use the SUBR function to return the value of an external subroutine. The SUBR function is commonly used in I-descriptors.

name is an expression that evaluates to the name of the subroutine to be executed. This subroutine must be cataloged in either a local catalog or the system catalog, or it must be a record in the same object file as the calling program. If *name* evaluates to the null value, the SUBR function fails, the program terminates, and a run-time error message appears.

argument is an expression evaluating to a variable name whose value is passed to the subroutine. You can pass up to 254 variables to the subroutine.

Subroutines called by the SUBR function must have a special syntax. The SUBROUTINE statement defining the subroutine must specify a dummy variable as the first parameter. The value of the subroutine is the value of the dummy variable when the subroutine finishes execution. Because the SUBROUTINE statement has this dummy parameter, the SUBR function must specify one argument less than the number of parameters in the SUBROUTINE statement. In other words, the SUBR function does not pass any argument to the subroutine through the first dummy parameter. The first argument passed by the SUBR function is referenced in the subroutine by the second parameter in the SUBROUTINE statement, and so on.

Example

The following example uses the globally cataloged subroutine *TEST:

```
OPEN "","SUN.MEMBER" TO FILE ELSE STOP "CAN'T OPEN DD"
EXECUTE "SELECT SUN.MEMBER"
10*
READNEXT KEY ELSE STOP
READ ITEM FROM FILE,KEY ELSE GOTO 10
X=ITEM<7>  ;* attribute 7 of file contains year
Z=SUBR("*TEST",X)
PRINT "YEARS=", Z
GOTO 10
```

This is the subroutine TEST:

```
SUBROUTINE TEST(RESULT,X)
DATE=OCONV(DATE(),"D2/")
YR=FIELD(DATE,'/',3)
YR='19':YR
RESULT=YR-X
RETURN
```

This is the program output:

```
15 records selected to Select List #0
YEARS=  3
YEARS=  5
YEARS=  2
YEARS=  6
YEARS=  1
YEARS=  0
YEARS=  0
YEARS=  1
YEARS=  4
YEARS=  6
YEARS=  1
YEARS=  2
YEARS=  7
YEARS=  1
YEARS=  0
```

SUBROUTINE statement

Syntax

SUBROUTINE [*name*] [([MAT] *variable* [, [MAT] *variable* ...])]

Description

Use the SUBROUTINE statement to identify an external subroutine. The SUBROUTINE statement must be the first noncomment line in the subroutine. Each external subroutine can contain only one SUBROUTINE statement.

An external subroutine is a separate program or set of statements that can be executed by other programs or subroutines (called *calling programs*) to perform a task. The external subroutine must be compiled and cataloged before another program can call it.

The SUBROUTINE statement can specify a subroutine name for documentation purposes; it need not be the same as the program name or the name by which it is called. The CALL statement must reference the subroutine by its name in the catalog, in the VOC file, or in the object file.

variables are variable names used in the subroutine to pass values between the calling programs and the subroutine. To pass an array, you must precede the array name with the keyword MAT. When an external subroutine is called, the CALL statement must specify the same number of variables as are specified in the SUBROUTINE statement. See the CALL statement in Volume 1 for more information.

Example

The following SUBROUTINE statements specify three variables, EM, GROSS, and TAX, the values of which are passed to the subroutine by the calling program:

```
SUBROUTINE ALONE(EM, GROSS, TAX)

SUBROUTINE STATE(EM,GROSS,TAX)
```

Syntax

SUBS (*array1*, *array2*)

CALL –SUBS (*return.array*, *array1*, *array2*)

CALL !SUBS (*return.array*, *array1*, *array2*)

Description

Use the SUBS function to create a dynamic array of the element-by-element subtraction of two dynamic arrays.

Each element of *array2* is subtracted from the corresponding element of *array1* with the result being returned in the corresponding element of a new dynamic array.

If an element of one dynamic array has no corresponding element in the other dynamic array, the missing element is evaluated as 0. If either of a corresponding pair of elements is the null value, null is returned for that element.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A=2:@VM:4:@VM:6:@SM:18
B=1:@VM:2:@VM:3:@VM:9
PRINT SUBS(A,B)
```

This is the program output:

```
1v2v3s18v-9
```

SUBSTRINGS function

Syntax

SUBSTRINGS (*dynamic.array*, *start*, *length*)

CALL –SUBSTRINGS (*return.array*, *dynamic.array*, *start*, *length*)

CALL !SUBSTRINGS (*return.array*, *dynamic.array*, *start*, *length*)

Description

Use the SUBSTRINGS function to create a dynamic array each of whose elements are substrings of the corresponding elements of *dynamic.array*.

start indicates the position of the first character of each element to be included in the substring. If *start* is 0 or a negative number, the starting position is assumed to be 1. If *start* is greater than the number of characters in the element, an empty string is returned.

length specifies the total length of the substring. If *length* is 0 or a negative number, an empty string is returned. If the sum of *start* and *length* is larger than the element, the substring ends with the last character of the element.

If an element of *dynamic.array* is the null value, null is returned for that element. If *start* or *length* evaluates to the null value, the SUBSTRINGS function fails, the program terminates, and a run-time error message appears.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array*.

Example

```
A="ABCDEF":@VM:"GH":@SM:"IJK"
PRINT SUBSTRINGS(A,3,2)
```

This is the program output:

```
CDvsK
```

Syntax

SUM (*dynamic.array*)

Description

Use the SUM function to calculate the sum of numeric data. Only elements at the lowest delimiter level of a dynamic array are summed. The total is returned as a single element at the next highest delimiter level.

The delimiters from highest to lowest are field, value, and subvalue.

There are seven levels of delimiters from CHAR(254) to CHAR(248): field mark, value mark, subvalue mark, text mark, CHAR(250), CHAR(249), and CHAR(248).

The SUM function removes the lowest delimiter level from a dynamic array. In a dynamic array that contains fields, values, and subvalues, the SUM function sums only the subvalues, returning the sums as values. In a dynamic array that contains fields and values, the SUM function sums only the values, returning the sums as fields. In a dynamic array that contains only fields, the SUM function sums the fields, returning the sum as the only field of the array. SUM functions can be applied repeatedly to raise multilevel data to the highest delimiter level or to a single value.

Nonnumeric values, except the null value, are treated as 0. If *dynamic.array* evaluates to the null value, null is returned. Any element that is the null value is ignored, unless all elements of *dynamic.array* are null, in which case null is returned.

Examples

In the following examples a field mark is shown by F, a value mark is shown by V, and a subvalue mark is shown by S.

Source Lines	Program Output
`X=20:@VM:18:@VM:9:@VM:30:@VM:80` `PRINT "SUM(X)=",SUM(X)`	`SUM(X)= 157`

`X=17:@FM:18:@FM:15`
`Y=10:@FM:20`
`PRINT "SUM(X)+SUM(Y)= ",SUM(X)+SUM(Y)`
	`SUM(X)+SUM(Y)=  80`

SUM function

Source Lines	Program Output

`X=3:@SM:4:@SM:10:@VM:3:@VM:20`
`Y=SUM(X)`
`PRINT "Y= ",Y`
`Z=SUM(Y)`
`PRINT "Z= ",Z`
	`Y=  17v3v20`
	`Z=  40`

Syntax

SUMMATION (*dynamic.array*)

CALL !SUMMATION (*result*, *array*)

Description

Use the SUMMATION function to return the sum of all the elements in *dynamic.array*. Nonnumeric values, except the null value, are treated as 0.

result is a variable containing the result of the sum.

dynamic.array is the dynamic array whose elements are to be added together.

Example

```
A=1:@VM:"ZERO":@SM:20:@FM:-25
PRINT "SUMMATION(A)=",SUMMATION(A)
```

This is the program output:

```
SUMMATION(A)=-4
```

SYSTEM function

Syntax

SYSTEM (*expression)*

Description

Use the SYSTEM function to check on the status of a system function.

expression evaluates to the number of the system function you want to check. If *expression* evaluates to the null value, the SYSTEM function fails, the program terminates, and a run-time error message appears.

The following table lists the values for *expression* and their meanings:

SYSTEM Function Values

Value	Action
1	Checks to see if the PRINTER ON statement has turned the printer on. Returns 1 if the printer is on and 0 if it is not.
2	Returns the page width as defined by the terminal characteristic settings.
3	Returns the page length as defined by the terminal characteristic settings.
4	Returns the number of lines remaining on the current page.
5	Returns the current page number.
6	Returns the current line number.
7	Returns the terminal code for the type of terminal the system believes you are using.
8,*n*	Checks whether the tape is attached. Returns the current block size if it is and –1 if it is not. *n* is the number of the tape unit. If it is not specified, tape unit 0 is assumed.
9	Returns the current CPU millisecond count.
10	Checks whether the DATA stack is active. Returns 1 if it is active and 0 if it is not.
11	Checks whether a select list is active. Returns 1 if a select list is active and 0 if one is not.

SYSTEM Function Values (Continued)

Value	Action
12	By default, returns the current system time in seconds (local time). If the TIME.MILLISECOND option is set (see $OPTIONS), returns the current system time in milliseconds.
13	Not used. Returns 0.
14	Not used. Returns 0.
15	Not used. Returns 0.
16	Returns 1 if running from a proc, otherwise returns 0.
17	Not used. Returns 0.
18	Returns the terminal number.
19	Returns the login name.
20	Not used. Returns 0.
21	Not used. Returns 0.
22	Not used. Returns 0.
23	Checks whether the **Break** key is enabled. Returns 1 if the **Break** key is enabled and 0 if it is not.
24	Checks whether character echoing is enabled. Returns 1 if character echoing is enabled and 0 if it is not.
25	Returns 1 if running from a phantom process, otherwise returns 0.
26	Returns the current prompt character.
27	Returns the user ID of the person using the routine.
28	Returns the effective user ID of the person using the routine. **Windows NT.** This is the same value as 27.
29	Returns the group ID of the person using the routine. **Windows NT.** This value is 0.
30	Returns the effective group ID of the person using the routine. **Windows NT.** This value is 0.
31	Returns the UniVerse serial number.
32	Returns the location of the UniVerse home directory.
33	Returns the last command on the command stack.

SYSTEM Function Values (Continued)

Value	Action
34	Returns data pending.
35	Returns the number of users currently in UniVerse.
36	Returns the maximum number of UniVerse users.
37	Returns the number of UNIX users; on Windows NT systems returns same value as 35.
38	Returns the pathname of the temporary directory.
42	Returns an empty string. On Windows NT systems returns the current value of the telnet client's IP address, or an empty string if the process evaluating the SYSTEM function is not the main UniVerse telnet process.
50	Returns the field number of the last READNEXT statement when reading an exploded select list.
60	Returns the current value of the UniVerse-configurable parameter TXMODE. The value can be either 1 or 0.
61	Returns the status of the transaction log daemon. 1 indicates the daemon is active; 0 indicates it is inactive.
91	Returns 0; on Windows NT, returns 1.
99	Returns the system time in the number of seconds since midnight Greenwich Mean Time (GMT), January 1, 1970.
1001	Returns the UniVerse flavor: 1 for IDEAL, 2 for PICK, 4 for INFORMATION, 8 for REALITY, 16 for IN2, and 64 for PIOPEN.
1017	Returns the user's supplementary UNIX groups in a dynamic array.
1021	Returns the GCI error number.
1200, *hostname*	Returns the UV/Net link number associated with *hostname*. If there is an internal error adding *hostname*, 0 returns. *hostname* is an expression that contains the host name from a file opened through UV/Net. It refers to the host name portion of the file's pathname. For example, in the pathname ORION*!/u1/filename*, *hostname* is ORION.

SYSTEM Function Values (Continued)

Value	Action
1201, *hostname*	Returns the RPC connection number associated with *hostname*. The UV/Net REMOTE.B interface program uses this number. If there is an internal error adding *hostname*, or if RPC has not yet opened, 0 returns. If the RPC connection was opened but is now closed, –1 returns.
1202, *hostname*	Returns the timeout associated with *hostname*. If there is no timeout associated with *hostname*, 0 returns.
1203	Returns the last RPC connection error number. This number is in the range 81000 through 81999. 81015 indicates that a timeout occurred. These error numbers correspond to error messages in the SYS.MESSAGE file.

Examples

The first example returns the number of lines left to print on a page, with the maximum defined by the TERM statement. The second example returns the current page number.

Source Lines	Program Output
`Q=4` `PRINT 'SYSTEM(Q)',SYSTEM(Q)`	`SYSTEM(Q)  20`
`PRINT 'X=',SYSTEM(5)`	`X=  0`

The next example sets a 30-second timeout for the UV/Net connection to the system ORION:

```
TIMEOUT SYSTEM(1200, "ORION"), 30
```

TABSTOP statement

Syntax

TABSTOP *expression*

Description

Use the TABSTOP statement to set the current tabstop width for PRINT statements. The initial tabstop setting is 10.

If *expression* evaluates to the null value, the TABSTOP statement fails, the program terminates, and a run-time error message appears.

Example

```
A="FIRST"
B="LAST"
PRINT A,B
TABSTOP 15
PRINT A,B
```

This is the program output:

```
FIRST  LAST
FIRST     LAST
```

Syntax

TAN (*expression*)

Description

Use the TAN function to return the trigonometric tangent of *expression*. *expression* represents an angle expressed in degrees.

Trying to take the tangent of a right angle results in a warning message, and a return value of 0. Numbers greater than 1E17 produce a warning message, and 0 is returned. If *expression* evaluates to the null value, null is returned.

Example

```
PRINT TAN(45)
```

This is the program output:

```
1
```

TANH function

Syntax

TANH (*expression*)

Description

Use the TANH function to return the hyperbolic tangent of *expression*. *expression* must be numeric and represents the angle expressed in degrees. If *expression* evaluates to the null value, null is returned.

Example

```
PRINT TANH(45)
```

This is the program output:

```
1
```

Syntax

TERMINFO (*argument*)

Description

Use the TERMINFO function to access the device-independent terminal handler string defined for the current terminal type. The TERMINFO function returns a dynamic array containing the terminal characteristics for the terminal type set by TERM or SET.TERM.TYPE.

argument can be 0 or 1, depending on whether the terminal characteristics are returned as stored, or converted to printable form. If *argument* is 0, the function returns the terminal characteristics in the form usable by BASIC applications for device-independent terminal handling with the TPARM and TPRINT statements. If *argument* is 1, the function returns characteristics in *terminfo* source format. Boolean values are returned as Y = true and N = false. The *terminfo* files contain many unprintable control characters that may adversely affect your terminal.

If *argument* evaluates to the null value, the TERMINFO function fails, the program terminates, and a run-timSe error message appears.

The easiest way to access the *terminfo* characteristics is by including the BASIC file UNIVERSE.INCLUDE TERMINFO in your program. The syntax is:

```
$INCLUDE UNIVERSE.INCLUDE TERMINFO
```

The file contains lines that equate each dynamic array element returned by TERMINFO with a name, so that each element can be easily accessed in your program. Once this file has been included in your program, you can use the defined names to access terminal characteristics. The following table lists the contents of this file:

TERMINFO EQUATEs

terminfo$ = terminfo(0)	
EQU TERMINAL.NAME	TO terminfo$<1>
EQU COLUMNS	TO terminfo$<2>
EQU LINES	TO terminfo$<3>
EQU CARRIAGE.RETURN	TO terminfo$<4>
EQU LINE.FEED	TO terminfo$<5>
EQU NEWLINE	TO terminfo$<6>
EQU BACKSPACE	TO terminfo$<7>

TERMINFO EQUATEs (Continued)

EQU BELL	TO terminfo$<8>
EQU SCREEN.FLASH	TO terminfo$<9>
EQU PADDING.CHARACTER	TO terminfo$<10>
EQU PAD.BAUD.RATE	TO terminfo$<11>
EQU HARD.COPY	TO terminfo$<12>
EQU OVERSTRIKES	TO terminfo$<13>
EQU ERASES.OVERSTRIKE	TO terminfo$<14>
EQU AUTOMATIC.RIGHT.MARGIN	TO terminfo$<15>
EQU RIGHT.MARGIN.EATS.NEWLINE	TO terminfo$<16>
EQU AUTOMATIC.LEFT.MARGIN	TO terminfo$<17>
EQU UNABLE.TO.PRINT.TILDE	TO terminfo$<18>
EQU ERASE.SCREEN	TO terminfo$<19>
EQU ERASE.TO.END.OF.SCREEN	TO terminfo$<20>
EQU ERASE.TO.BEGINNING.OF.SCREEN	TO terminfo$<21>
EQU ERASE.LINE	TO terminfo$<22>
EQU ERASE.TO.END.OF.LINE	TO terminfo$<23>
EQU ERASE.TO.BEGINNING.OF.LINE	TO terminfo$<24>
EQU ERASE.CHARACTERS	TO terminfo$<25>
EQU MOVE.CURSOR.TO.ADDRESS	TO terminfo$<26>
EQU MOVE.CURSOR.TO.COLUMN	TO terminfo$<27>
EQU MOVE.CURSOR.TO.ROW	TO terminfo$<28>
EQU MOVE.CURSOR.RIGHT	TO terminfo$<29>
EQU MOVE.CURSOR.LEFT	TO terminfo$<30>
EQU MOVE.CURSOR.DOWN	TO terminfo$<31>
EQU MOVE.CURSOR.UP	TO terminfo$<32>
EQU MOVE.CURSOR.RIGHT.PARM	TO terminfo$<33>
EQU MOVE.CURSOR.LEFT.PARM	TO terminfo$<34>
EQU MOVE.CURSOR.DOWN.PARM	TO terminfo$<35>
EQU MOVE.CURSOR.UP.PARM	TO terminfo$<36>
EQU MOVE.CURSOR.TO.HOME	TO terminfo$<37>
EQU MOVE.CURSOR.TO.LAST.LINE	TO terminfo$<38>
EQU CURSOR.SAVE	TO terminfo$<39>
EQU CURSOR.RESTORE	TO terminfo$<40>
EQU INSERT.CHARACTER	TO terminfo$<41>
EQU INSERT.CHARACTER.PARM	TO terminfo$<42>

TERMINFO EQUATEs (Continued)

EQU INSERT.MODE.BEGIN	TO terminfo$<43>
EQU INSERT.MODE.END	TO terminfo$<44>
EQU INSERT.PAD	TO terminfo$<45>
EQU MOVE.INSERT.MODE	TO terminfo$<46>
EQU INSERT.NULL.SPECIAL	TO terminfo$<47>
EQU DELETE.CHARACTER	TO terminfo$<48>
EQU DELETE.CHARACTER.PARM	TO terminfo$<49>
EQU INSERT.LINE	TO terminfo$<50>
EQU INSERT.LINE.PARM	TO terminfo$<51>
EQU DELETE.LINE	TO terminfo$<52>
EQU DELETE.LINE.PARM	TO terminfo$<53>
EQU SCROLL.UP	TO terminfo$<54>
EQU SCROLL.UP.PARM	TO terminfo$<55>
EQU SCROLL.DOWN	TOterminfo$<56>
EQU SCROLL.DOWN.PARM	TOterminfo$<57>
EQU CHANGE.SCROLL.REGION	TOterminfo$<58>
EQU SCROLL.MODE.END	TOterminfo$<59>
EQU SCROLL.MODE.BEGIN	TOterminfo$<60>
EQU VIDEO.NORMAL	TO terminfo$<61>
EQU VIDEO.REVERSE	TO terminfo$<62>
EQU VIDEO.BLINK	TO terminfo$<63>
EQU VIDEO.UNDERLINE	TO terminfo$<64>
EQU VIDEO.DIM	TO terminfo$<65>
EQU VIDEO.BOLD	TO terminfo$<66>
EQU VIDEO.BLANK	TO terminfo$<67>
EQU VIDEO.STANDOUT	TO terminfo$<68>
EQU VIDEO.SPACES	TO terminfo$<69>
EQU MOVE.VIDEO.MODE	TO terminfo$<70>
EQU TAB	TO terminfo$<71>
EQU BACK.TAB	TO terminfo$<72>
EQU TAB.STOP.SET	TO terminfo$<73>
EQU TAB.STOP.CLEAR	TO terminfo$<74>
EQU CLEAR.ALL.TAB.STOPS	TO terminfo$<75>
EQU TAB.STOP.INITIAL	TO terminfo$<76>
EQU WRITE.PROTECT.BEGIN	TO terminfo$<77>

TERMINFO EQUATEs (Continued)

EQU WRITE.PROTECT.END	TO terminfo$<78>
EQU SCREEN.PROTECT.BEGIN	TO terminfo$<79>
EQU SCREEN.PROTECT.END	TO terminfo$<80>
EQU WRITE.PROTECT.COLUMN	TO terminfo$<81>
EQU PROTECT.VIDEO.NORMAL	TO terminfo$<82>
EQU PROTECT.VIDEO.REVERSE	TO terminfo$<83>
EQU PROTECT.VIDEO.BLINK	TO terminfo$<84>
EQU PROTECT.VIDEO.UNDERLINE	TO terminfo$<85>
EQU PROTECT.VIDEO.DIM	TO terminfo$<86>
EQU PROTECT.VIDEO.BOLD	TO terminfo$<87>
EQU PROTECT.VIDEO.BLANK	TO terminfo$<88>
EQU PROTECT.VIDEO.STANDOUT	TO terminfo$<89>
EQU BLOCK.MODE.BEGIN	TO terminfo$<90>
EQU BLOCK.MODE.END	TO terminfo$<91>
EQU SEND.LINE.ALL	TO terminfo$<92>
EQU SEND.LINE.UNPROTECTED	TO terminfo$<93>
EQU SEND.PAGE.ALL	TO terminfo$<94>
EQU SEND.PAGE.UNPROTECTED	TO terminfo$<95>
EQU SEND.MESSAGE.ALL	TO terminfo$<96>
EQU SEND.MESSAGE.UNPROTECTED	TO terminfo$<97>
EQU TERMINATE.FIELD	TO terminfo$<98>
EQU TERMINATE.LINE	TO terminfo$<99>
EQU TERMINATE.PAGE	TO terminfo$<100>
EQU STORE.START.OF.MESSAGE	TO terminfo$<101>
EQU STORE.END.OF.MESSAGE	TO terminfo$<102>
EQU LINEDRAW.BEGIN	TO terminfo$<103>
EQU LINEDRAW.END	TO terminfo$<104>
EQU MOVE.LINEDRAW.MODE	TO terminfo$<105>
EQU LINEDRAW.CHARACTER	TO terminfo$<106>
EQU LINEDRAW.UPPER.LEFT.CORNER	TO terminfo$<107>
EQU LINEDRAW.UPPER.RIGHT.CORNER	TO terminfo$<108>
EQU LINEDRAW.LOWER.LEFT.CORNER	TO terminfo$<109>
EQU LINEDRAW.LOWER.RIGHT.CORNER	TO terminfo$<110>
EQU LINEDRAW.LEFT.VERTICAL	TO terminfo$<111>
EQU LINEDRAW.CENTER.VERTICAL	TO terminfo$<112>

TERMINFO EQUATEs (Continued)

EQU LINEDRAW.RIGHT.VERTICAL	TO terminfo$<113>
EQU LINEDRAW.UPPER.HORIZONTAL	TO terminfo$<114>
EQU LINEDRAW.CENTER.HORIZONTAL	TO terminfo$<115>
EQU LINEDRAW.LOWER.HORIZONTAL	TO terminfo$<116>
EQU LINEDRAW.UPPER.TEE	TO terminfo$<117>
EQU LINEDRAW.LOWER.TEE	TO terminfo$<118>
EQU LINEDRAW.LEFT.TEE	TO terminfo$<119>
EQU LINEDRAW.RIGHT.TEE	TO terminfo$<120>
EQU LINEDRAW.CROSS	TO terminfo$<121>
EQU CURSOR.NORMAL	TO terminfo$<122>
EQU CURSOR.VISIBLE	TO terminfo$<123>
EQU CURSOR.INVISIBLE	TO terminfo$<124>
EQU SCREEN.VIDEO.ON	TO terminfo$<125>
EQU SCREEN.VIDEO.OFF	TO terminfo$<126>
EQU KEYCLICK.ON	TO terminfo$<127>
EQU KEYCLICK.OFF	TO terminfo$<128>
EQU KEYBOARD.LOCK.ON	TO terminfo$<129>
EQU KEYBOARD.LOCK.OFF	TO terminfo$<130>
EQU MONITOR.MODE.ON	TO terminfo$<131>
EQU MONITOR.MODE.OFF	TO terminfo$<132>
EQU PRINT.SCREEN	TO terminfo$<133>
EQU PRINT.MODE.BEGIN	TO terminfo$<134>
EQU PRINT.MODE.END	TO terminfo$<135>
EQU HAS.STATUS.LINE	TO terminfo$<136>
EQU STATUS.LINE.WIDTH	TO terminfo$<137>
EQU STATUS.LINE.BEGIN	TO terminfo$<138>
EQU STATUS.LINE.END	TO terminfo$<139>
EQU STATUS.LINE.DISABLE	TO terminfo$<140>
EQU HAS.FUNCTION.LINE	TO terminfo$<141>
EQU FUNCTION.LINE.BEGIN	TO terminfo$<142>
EQU FUNCTION.LINE.END	TO terminfo$<143>
EQU KEY.BACKSPACE	TO terminfo$<144>
EQU KEY.MOVE.CURSOR.RIGHT	TO terminfo$<145>
EQU KEY.MOVE.CURSOR.LEFT	TO terminfo$<146>
EQU KEY.MOVE.CURSOR.DOWN	TO terminfo$<147>

TERMINFO EQUATEs (Continued)

EQU KEY.MOVE.CURSOR.UP	TO terminfo$<148>
EQU KEY.MOVE.CURSOR.TO.HOME	TO terminfo$<149>
EQU KEY.MOVE.CURSOR.TO.LAST.LINE	TO terminfo$<150>
EQU KEY.INSERT.CHARACTER	TO terminfo$<151>
EQU KEY.INSERT.MODE.ON	TO terminfo$<152>
EQU KEY.INSERT.MODE.END	TO terminfo$<153>
EQU KEY.INSERT.MODE.TOGGLE	TO terminfo$<154>
EQU KEY.DELETE.CHARACTER	TO terminfo$<155>
EQU KEY.INSERT.LINE	TO terminfo$<156>
EQU KEY.DELETE.LINE	TO terminfo$<157>
EQU KEY.ERASE.SCREEN	TO terminfo$<158>
EQU KEY.ERASE.END.OF.LINE	TO terminfo$<159>
EQU KEY.ERASE.END.OF.SCREEN	TO terminfo$<160>
EQU KEY.BACK.TAB	TO terminfo$<161>
EQU KEY.TAB.STOP.SET	TO terminfo$<162>
EQU KEY.TAB.STOP.CLEAR	TO terminfo$<163>
EQU KEY.TAB.STOP.CLEAR.ALL	TO terminfo$<164>
EQU KEY.NEXT.PAGE	TO terminfo$<165>
EQU KEY.PREVIOUS.PAGE	TO terminfo$<166>
EQU KEY.SCROLL.UP	TO terminfo$<167>
EQU KEY.SCROLL.DOWN	TO terminfo$<168>
EQU KEY.SEND.DATA	TO terminfo$<169>
EQU KEY.PRINT	TO terminfo$<170>
EQU KEY.FUNCTION.0	TO terminfo$<171>
EQU KEY.FUNCTION.1	TO terminfo$<172>
EQU KEY.FUNCTION.2	TO terminfo$<173>
EQU KEY.FUNCTION.3	TO terminfo$<174>
EQU KEY.FUNCTION.4	TO terminfo$<175>
EQU KEY.FUNCTION.5	TO terminfo$<176>
EQU KEY.FUNCTION.6	TO terminfo$<177>
EQU KEY.FUNCTION.7	TO terminfo$<178>
EQU KEY.FUNCTION.8	TO terminfo$<179>
EQU KEY.FUNCTION.9	TO terminfo$<180>
EQU KEY.FUNCTION.10	TO terminfo$<181>
EQU KEY.FUNCTION.11	TO terminfo$<182>

TERMINFO EQUATEs (Continued)

EQU KEY.FUNCTION.12	TO terminfo$<183>
EQU KEY.FUNCTION.13	TO terminfo$<184>
EQU KEY.FUNCTION.14	TO terminfo$<185>
EQU KEY.FUNCTION.15	TO terminfo$<186>
EQU KEY.FUNCTION.16	TO terminfo$<187>
EQU LABEL.KEY.FUNCTION.0	TO terminfo$<188>
EQU LABEL.KEY.FUNCTION.1	TO terminfo$<189>
EQU LABEL.KEY.FUNCTION.2	TO terminfo$<190>
EQU LABEL.KEY.FUNCTION.3	TO terminfo$<191>
EQU LABEL.KEY.FUNCTION.4	TO terminfo$<192>
EQU LABEL.KEY.FUNCTION.5	TO terminfo$<193>
EQU LABEL.KEY.FUNCTION.6	TO terminfo$<194>
EQU LABEL.KEY.FUNCTION.7	TO terminfo$<195>
EQU LABEL.KEY.FUNCTION.8	TO terminfo$<196>
EQU LABEL.KEY.FUNCTION.9	TO terminfo$<197>
EQU LABEL.KEY.FUNCTION.10	TO terminfo$<198>
EQU LABEL.KEY.FUNCTION.11	TO terminfo$<199>
EQU LABEL.KEY.FUNCTION.12	TO terminfo$<200>
EQU LABEL.KEY.FUNCTION.13	TO terminfo$<201>
EQU LABEL.KEY.FUNCTION.14	TO terminfo$<202>
EQU LABEL.KEY.FUNCTION.15	TO terminfo$<203>
EQU LABEL.KEY.FUNCTION.16	TO terminfo$<204>
EQU KEYEDIT.FUNCTION	TO terminfo$<205>
EQU KEYEDIT.ESCAPE	TO terminfo$<206>
EQU KEYEDIT.EXIT	TO terminfo$<207>
EQU KEYEDIT.BACKSPACE	TO terminfo$<208>
EQU KEYEDIT.MOVE.BACKWARD	TO terminfo$<209>
EQU KEYEDIT.MOVE.FORWARD	TO terminfo$<210>
EQU KEYEDIT.INSERT.CHARACTER	TO terminfo$<211>
EQU KEYEDIT.INSERT.MODE.BEGIN	TO terminfo$<212>
EQU KEYEDIT.INSERT.MODE.END	TO terminfo$<213>
EQU KEYEDIT.INSERT.MODE.TOGGLE	TO terminfo$<214>
EQU KEYEDIT.DELETE.CHARACTER	TO terminfo$<215>
EQU KEYEDIT.ERASE.END.OF.FIELD	TO terminfo$<216>
EQU KEYEDIT.ERASE.FIELD	TO terminfo$<217>

TERMINFO EQUATEs (Continued)

EQU AT.NEGATIVE.1	TO terminfo$<218>
EQU AT.NEGATIVE.2	TO terminfo$<219>
EQU AT.NEGATIVE.3	TO terminfo$<220>
EQU AT.NEGATIVE.4	TO terminfo$<221>
EQU AT.NEGATIVE.5	TO terminfo$<222>
EQU AT.NEGATIVE.6	TO terminfo$<223>
EQU AT.NEGATIVE.7	TO terminfo$<224>
EQU AT.NEGATIVE.8	TO terminfo$<225>
EQU AT.NEGATIVE.9	TO terminfo$<226>
EQU AT.NEGATIVE.10	TO terminfo$<227>
EQU AT.NEGATIVE.11	TO terminfo$<228>
EQU AT.NEGATIVE.12	TO terminfo$<229>
EQU AT.NEGATIVE.13	TO terminfo$<230>
EQU AT.NEGATIVE.14	TO terminfo$<231>
EQU AT.NEGATIVE.15	TO terminfo$<232>
EQU AT.NEGATIVE.16	TO terminfo$<233>
EQU AT.NEGATIVE.17	TO terminfo$<234>
EQU AT.NEGATIVE.18	TO terminfo$<235>
EQU AT.NEGATIVE.19	TO terminfo$<236>
EQU AT.NEGATIVE.20	TO terminfo$<237>
EQU AT.NEGATIVE.21	TO terminfo$<238>
EQU AT.NEGATIVE.22	TO terminfo$<239>
EQU AT.NEGATIVE.23	TO terminfo$<240>
EQU AT.NEGATIVE.24	TO terminfo$<241>
EQU AT.NEGATIVE.25	TO terminfo$<242>
EQU AT.NEGATIVE.26	TO terminfo$<243>
EQU AT.NEGATIVE.27	TO terminfo$<244>
EQU AT.NEGATIVE.28	TO terminfo$<245>
EQU AT.NEGATIVE.29	TO terminfo$<246>
EQU AT.NEGATIVE.30	TO terminfo$<247>
EQU AT.NEGATIVE.31	TO terminfo$<248>
EQU AT.NEGATIVE.32	TO terminfo$<249>
EQU AT.NEGATIVE.33	TO terminfo$<250>
EQU AT.NEGATIVE.34	TO terminfo$<251>
EQU AT.NEGATIVE.35	TO terminfo$<252>

TERMINFO EQUATEs (Continued)

EQU AT.NEGATIVE.36	TO terminfo$<253>
EQU AT.NEGATIVE.37	TO terminfo$<254>
EQU AT.NEGATIVE.38	TO terminfo$<255>
EQU AT.NEGATIVE.39	TO terminfo$<256>
EQU AT.NEGATIVE.40	TO terminfo$<257>
EQU AT.NEGATIVE.41	TO terminfo$<258>
EQU AT.NEGATIVE.42	TO terminfo$<259>
EQU AT.NEGATIVE.43	TO terminfo$<260>
EQU AT.NEGATIVE.44	TO terminfo$<261>
EQU AT.NEGATIVE.45	TO terminfo$<262>
EQU AT.NEGATIVE.46	TO terminfo$<263>
EQU AT.NEGATIVE.47	TO terminfo$<264>
EQU AT.NEGATIVE.48	TO terminfo$<265>
EQU AT.NEGATIVE.49	TO terminfo$<266>
EQU AT.NEGATIVE.50	TO terminfo$<267>
EQU AT.NEGATIVE.51	TO terminfo$<268>
EQU AT.NEGATIVE.52	TO terminfo$<269>
EQU AT.NEGATIVE.53	TO terminfo$<270>
EQU AT.NEGATIVE.54	TO terminfo$<271>
EQU AT.NEGATIVE.55	TO terminfo$<272>
EQU AT.NEGATIVE.56	TO terminfo$<273>
EQU AT.NEGATIVE.57	TO terminfo$<274>
EQU AT.NEGATIVE.58	TO terminfo$<275>
EQU AT.NEGATIVE.59	TO terminfo$<276>
EQU AT.NEGATIVE.60	TO terminfo$<277>
EQU AT.NEGATIVE.61	TO terminfo$<278>
EQU AT.NEGATIVE.62	TO terminfo$<279>
EQU AT.NEGATIVE.63	TO terminfo$<280>
EQU AT.NEGATIVE.64	TO terminfo$<281>
EQU AT.NEGATIVE.65	TO terminfo$<282>
EQU AT.NEGATIVE.66	TO terminfo$<283>
EQUAT.NEGATIVE.67	TO terminfo$<284>
EQUAT.NEGATIVE.68	TO terminfo$<285>
EQU AT.NEGATIVE.69	TO terminfo$<286>
EQU AT.NEGATIVE.70	TO terminfo$<287>

TERMINFO EQUATEs (Continued)

EQU AT.NEGATIVE.71	TO terminfo$<288>
EQU AT.NEGATIVE.72	TO terminfo$<289>
EQU AT.NEGATIVE.73	TO terminfo$<290>
EQU AT.NEGATIVE.74	TO terminfo$<291>
EQU AT.NEGATIVE.75	TO terminfo$<292>
EQU AT.NEGATIVE.76	TO terminfo$<293>
EQU AT.NEGATIVE.77	TO terminfo$<294>
EQUAT.NEGATIVE.78	TO terminfo$<295>
EQU AT.NEGATIVE.79	TO terminfo$<296>
EQU AT.NEGATIVE.80	TO terminfo$<297>
EQU AT.NEGATIVE.81	TO terminfo$<298>
EQU AT.NEGATIVE.82	TO terminfo$<299>
EQU AT.NEGATIVE.83	TO terminfo$<300>
EQU AT.NEGATIVE.84	TO terminfo$<301>
EQU AT.NEGATIVE.85	TO terminfo$<302>
EQU AT.NEGATIVE.86	TO terminfo$<303>
EQU AT.NEGATIVE.87	TO terminfo$<304>
EQU AT.NEGATIVE.88	TO terminfo$<305>
EQU AT.NEGATIVE.89	TO terminfo$<306>
EQU AT.NEGATIVE.90	TO terminfo$<307>
EQU AT.NEGATIVE.91	TO terminfo$<308>
EQU AT.NEGATIVE.92	TO terminfo$<309>
EQU AT.NEGATIVE.93	TO terminfo$<310>
EQU AT.NEGATIVE.94	TO terminfo$<311>
EQU AT.NEGATIVE.95	TO terminfo$<312>
EQU AT.NEGATIVE.96	TO terminfo$<313>
EQU AT.NEGATIVE.97	TO terminfo$<314>
EQU AT.NEGATIVE.98	TO terminfo$<315>
EQU AT.NEGATIVE.99	TO terminfo$<316>
EQU AT.NEGATIVE.100	TO terminfo$<317>
EQU AT.NEGATIVE.101	TO terminfo$<318>
EQU AT.NEGATIVE.102	TO terminfo$<319>
EQU AT.NEGATIVE.103	TO terminfo$<320>
EQU AT.NEGATIVE.104	TO terminfo$<321>
EQU AT.NEGATIVE.105	TO terminfo$<322>

TERMINFO EQUATEs (Continued)

EQU AT.NEGATIVE.106	TO terminfo$<323>
EQU AT.NEGATIVE.107	TO terminfo$<324>
EQU AT.NEGATIVE.108	TO terminfo$<325>
EQU AT.NEGATIVE.109	TO terminfo$<326>
EQU AT.NEGATIVE.110	TO terminfo$<327>
EQU AT.NEGATIVE.111	TO terminfo$<328>
EQU AT.NEGATIVE.112	TO terminfo$<329>
EQU AT.NEGATIVE.113	TO terminfo$<330>
EQU AT.NEGATIVE.114	TO terminfo$<331>
EQU AT.NEGATIVE.115	TO terminfo$<332>
EQU AT.NEGATIVE.116	TO terminfo$<333>
EQU AT.NEGATIVE.117	TO terminfo$<334>
EQU AT.NEGATIVE.118	TO terminfo$<335>
EQU AT.NEGATIVE.119	TO terminfo$<336>
EQU AT.NEGATIVE.120	TO terminfo$<337>
EQU AT.NEGATIVE.121	TO terminfo$<338>
EQU AT.NEGATIVE.122	TO terminfo$<339>
EQU AT.NEGATIVE.123	TO terminfo$<340>
EQU AT.NEGATIVE.124	TO terminfo$<341>
EQU AT.NEGATIVE.125	TO terminfo$<342>
EQU AT.NEGATIVE.126	TO terminfo$<343>
EQU AT.NEGATIVE.127	TO terminfo$<344>
EQU AT.NEGATIVE.128	TO terminfo$<345>
EQU DBLE.LDRAW.UP.LEFT.CORNER	TO terminfo$<379>
EQU DBLE.LDRAW.UP.RIGHT.CORNER	TO terminfo$<380>
EQU DBLE.LDRAW.LO.LEFT.CORNER	TO terminfo$<381>
EQU DBLE.LDRAW.LO.RIGHT.CORNER	TO terminfo$<382>
EQU DBLE.LDRAW.HORIZ	TO terminfo$<383>
EQU DBLE.LDRAW.VERT	TO terminfo$<384>
EQU DBLE.LDRAW.UP.TEE	TO terminfo$<385>
EQU DBLE.LDRAW.LO.TEE	TO terminfo$<386>
EQU DBLE.LDRAW.LEFT.TEE	TO terminfo$<387>
EQU DBLE.LDRAW.RIGHT.TEE	TO terminfo$<388>
EQU DBLE.LDRAW.CROSS	TO terminfo$<389>
EQU LDRAW.LEFT.TEE.DBLE.HORIZ	TO terminfo$<390>

TERMINFO EQUATEs (Continued)

EQU LDRAW.LEFT.TEE.DBLE.VERT	TO terminfo$<391>
EQU LDRAW.RIGHT.TEE.DBLE.HORIZ	TO terminfo$<392>
EQU LDRAW.RIGHT.TEE.DBLE.VERT	TO terminfo$<393>
EQU LDRAW.LOWER.TEE.DBLE.HORIZ	TO terminfo$<394>
EQU LDRAW.LOWER.TEE.DBLE.VERT	TO terminfo$<395>
EQU LDRAW.UP.TEE.DBLE.HORIZ	TO terminfo$<396>
EQU LDRAW.UP.TEE.DBLE.VERT	TO terminfo$<397>
EQU LDRAW.UP.LEFT.CORNER.DBLE.HORIZ	TO terminfo$<398>
EQU LDRAW.UP.LEFT.CORNER.DBLE.VERT	TO terminfo$<399>
EQU LDRAW.UP.RIGHT.CORNER.DBLE.HORIZ	TO terminfo$<400>
EQU LDRAW.UP.RIGHT.CORNER.DBLE.VERT	TO terminfo$<401>
EQU LDRAW.LO.LEFT.CORNER.DBLE.HORIZ	TO terminfo$<402>
EQU LDRAW.LO.LEFT.CORNER.DBLE.VERT	TO terminfo$<403>
EQU LDRAW.LO.RIGHT.CORNER.DBLE.HORIZ	TO terminfo$<404>
EQU LDRAW.LO.RIGHT.CORNER.DBLE.VERT	TO terminfo$<405>
EQU LDRAW.CROSS.DBLE.HORIZ	TO terminfo$<406>
EQU LDRAW.CROSS.DBLE.VERT	TO terminfo$<407>
EQU NO.ESC.CTLC	TO terminfo$<408>
EQU CEOL.STANDOUT.GLITCH	TO terminfo$<409>
EQU GENERIC.TYPE	TO terminfo$<410>
EQU HAS.META.KEY	TO terminfo$<411>
EQU MEMORY.ABOVE	TO terminfo$<412>
EQU MEMORY.BELOW	TO terminfo$<413>
EQU STATUS.LINE.ESC.OK	TO terminfo$<414>
EQU DEST.TABS.MAGIC.SMSO	TO terminfo$<415>
EQU TRANSPARENT.UNDERLINE	TO terminfo$<416>
EQU XON.XOFF	TO terminfo$<417>
EQU NEEDS.XON.XOFF	TO terminfo$<418>
EQU PRTR.SILENT	TO terminfo$<419>
EQU HARD.CURSOR	TO terminfo$<420>
EQU NON.REV.RMCUP	TO terminfo$<421>
EQU NO.PAD.CHAR	TO terminfo$<422>
EQU LINES.OF.MEMORY	TO terminfo$<423>
EQU VIRTUAL.TERMINAL	TO terminfo$<424>
EQU NUM.LABELS	TO terminfo$<425>

TERMINFO EQUATEs (Continued)

EQU LABEL.HEIGHT	TO terminfo$<426>
EQU LABEL.WIDTH	TO terminfo$<427>
EQU LINE.ATTRIBUTE	TO terminfo$<428>
EQU COMMAND.CHARACTER	TO terminfo$<429>
EQU CURSOR.MEM.ADDRESS	TO terminfo$<430>
EQU DOWN.HALF.LINE	TO terminfo$<431>
EQU ENTER.CA.MODE	TO terminfo$<432>
EQU ENTER.DELETE.MODE	TO terminfo$<433>
EQU ENTER.PROTECTED.MODE	TO terminfo$<434>
EQU EXIT.ATTRIBUTE.MODE	TO terminfo$<435>
EQU EXIT.CA.MODE	TO terminfo$<436>
EQU EXIT.DELETE.MODE	TO terminfo$<437>
EQU EXIT.STANDOUT.MODE	TO terminfo$<438>
EQU EXIT.UNDERLINE.MODE	TO terminfo$<439>
EQU FORM.FEED	TO terminfo$<440>
EQU INIT.1STRING	TO terminfo$<441>
EQU INIT.2STRING	TO terminfo$<442>
EQU INIT.3STRING	TO terminfo$<443>
EQU INIT.FILE	TO terminfo$<444>
EQU INS.PREFIX	TO terminfo$<445>
EQU KEY.IC	TO terminfo$<446>
EQU KEYPAD.LOCAL	TO terminfo$<447>
EQU KEYPAD.XMIT	TO terminfo$<448>
EQU META.OFF	TO terminfo$<449>
EQU META.ON	TO terminfo$<450>
EQU PKEY.KEY	TO terminfo$<451>
EQU PKEY.LOCAL	TO terminfo$<452>
EQU PKEY.XMIT	TO terminfo$<453>
EQU REPEAT.CHAR	TO terminfo$<454>
EQU RESET.1STRING	TO terminfo$<455>
EQU RESET.2STRING	TO terminfo$<456>
EQU RESET.3STRING	TO terminfo$<457>
EQU RESET.FILE	TO terminfo$<458>
EQU SET.ATTRIBUTES	TO terminfo$<459>
EQU SET.WINDOW	TO terminfo$<460>

TERMINFO EQUATEs (Continued)

EQU UNDERLINE.CHAR	TO terminfo$<461>
EQU UP.HALF.LINE	TO terminfo$<462>
EQU INIT.PROG	TO terminfo$<463>
EQU KEY.A1	TO terminfo$<464>
EQU KEY.A3	TO terminfo$<465>
EQU KEY.B2	TO terminfo$<466>
EQU KEY.C1	TO terminfo$<467>
EQU KEY.C3	TO terminfo$<468>
EQU PRTR.NON	TO terminfo$<469>
EQU CHAR.PADDING	TO terminfo$<470>
EQU LINEDRAW.CHARS	TO terminfo$<471>
EQU PLAB.NORM	TO terminfo$<472>
EQU ENTER.XON.MODE	TO terminfo$<473>
EQU EXIT.XON.MODE	TO terminfo$<474>
EQU ENTER.AM.MODE	TO terminfo$<475>
EQU EXIT.AM.MODE	TO terminfo$<476>
EQU XON.CHARACTER	TO terminfo$<477>
EQU XOFF.CHARACTER	TO terminfo$<478>
EQU ENABLE.LINEDRAW	TO terminfo$<479>
EQU LABEL.ON	TO terminfo$<480>
EQU LABEL.OFF	TO terminfo$<481>
EQU KEY.BEG	TO terminfo$<482>
EQU KEY.CANCEL	TO terminfo$<483>
EQU KEY.CLOSE	TO terminfo$<484>
EQU KEY.COMMAND	TO terminfo$<485>
EQU KEY.COPY	TO terminfo$<486>
EQU KEY.CREATE	TO terminfo$<487>
EQU KEY.END	TO terminfo$<488>
EQU KEY.ENTER	TO terminfo$<489>
EQU KEY.EXIT	TO terminfo$<490>
EQU KEY.FIND	TO terminfo$<491>
EQU KEY.HELP	TO terminfo$<492>
EQU KEY.MARK	TO terminfo$<493>
EQU KEY.MESSAGE	TO terminfo$<494>
EQU KEY.MOVE	TO terminfo$<495>

TERMINFO EQUATEs (Continued)

EQU KEY.NEXT	TO terminfo$<496>
EQU KEY.OPEN	TO terminfo$<497>
EQU KEY.OPTIONS	TO terminfo$<498>
EQU KEY.PREVIOUS	TO terminfo$<499>
EQU KEY.REDO	TO terminfo$<500>
EQU KEY.REFERENCE	TO terminfo$<501>
EQU KEY.REFRESH	TO terminfo$<502>
EQU KEY.REPLACE	TO terminfo$<503>
EQU KEY.RESTART	TO terminfo$<504>
EQU KEY.RESUME	TO terminfo$<505>
EQU KEY.SAVE	TO terminfo$<506>
EQU KEY.SUSPEND	TO terminfo$<507>
EQU KEY.UNDO	TO terminfo$<508>
EQU KEY.SBEG	TO terminfo$<509>
EQU KEY.SCANCEL	TO terminfo$<510>
EQU KEY.SCOMMAND	TO terminfo$<511>
EQU KEY.SCOPY	TO terminfo$<512>
EQU KEY.SCREATE	TO terminfo$<513>
EQU KEY.SDC	TO terminfo$<514>
EQU KEY.SDL	TO terminfo$<515>
EQU KEY.SELECT	TO terminfo$<516>
EQU KEY.SEND	TO terminfo$<517>
EQU KEY.SEOL	TO terminfo$<518>
EQU KEY.SEXIT	TO terminfo$<519>
EQU KEY.SFIND	TO terminfo$<520>
EQU KEY.SHELP	TO terminfo$<521>
EQU KEY.SHOME	TO terminfo$<522>
EQU KEY.SIC	TO terminfo$<523>
EQU KEY.SLEFT	TO terminfo$<524>
EQU KEY.SMESSAGE	TO terminfo$<525>
EQU KEY.SMOVE	TO terminfo$<526>
EQU KEY.SNEXT	TO terminfo$<527>
EQU KEY.SOPTIONS	TO terminfo$<528>
EQU KEY.SPREVIOUS	TO terminfo$<529>
EQU KEY.SPRINT	TO terminfo$<530>

TERMINFO EQUATEs (Continued)

EQU KEY.SREDO	TO terminfo$<531>
EQU KEY.SREPLACE	TO terminfo$<532>
EQU KEY.SRIGHT	TO terminfo$<533>
EQU KEY.SRESUM	TO terminfo$<534>
EQU KEY.SSAVE	TO terminfo$<535>
EQU KEY.SSUSPEND	TO terminfo$<536>
EQU KEY.SUNDO	TO terminfo$<537>
EQU REQ.FOR.INPUT	TO terminfo$<538>
EQU KEY.F17	TO terminfo$<539>
EQU KEY.F18	TO terminfo$<540>
EQU KEY.F19	TO terminfo$<541>
EQU KEY.F20	TO terminfo$<542>
EQU KEY.F21	TO terminfo$<543>
EQU KEY.F22	TO terminfo$<544>
EQU KEY.F23	TO terminfo$<545>
EQU KEY.F24	TO terminfo$<546>
EQU KEY.F25	TO terminfo$<547>
EQU KEY.F26	TO terminfo$<548>
EQU KEY.F27	TO terminfo$<549>
EQU KEY.F28	TO terminfo$<550>
EQU KEY.F29	TO terminfo$<551>
EQU KEY.F30	TO terminfo$<552>
EQU KEY.F31	TO terminfo$<553>
EQU KEY.F32	TO terminfo$<554>
EQU KEY.F33	TO terminfo$<555>
EQU KEY.F34	TO terminfo$<556>
EQU KEY.F35	TO terminfo$<557>
EQU KEY.F36	TO terminfo$<558>
EQU KEY.F37	TO terminfo$<559>
EQU KEY.F38	TO terminfo$<560>
EQU KEY.F39	TO terminfo$<561>
EQU KEY.F40	TO terminfo$<562>
EQU KEY.F41	TO terminfo$<563>
EQU KEY.F42	TO terminfo$<564>
EQU KEY.F43	TO terminfo$<565>

TERMINFO EQUATEs (Continued)

EQU KEY.F44	TO terminfo$<566>
EQU KEY.F45	TO terminfo$<567>
EQU KEY.F46	TO terminfo$<568>
EQU KEY.F47	TO terminfo$<569>
EQU KEY.F48	TO terminfo$<570>
EQU KEY.F49	TO terminfo$<571>
EQU KEY.F50	TO terminfo$<572>
EQU KEY.F51	TO terminfo$<573>
EQU KEY.F52	TO terminfo$<574>
EQU KEY.F53	TO terminfo$<575>
EQU KEY.F54	TO terminfo$<576>
EQU KEY.F55	TO terminfo$<577>
EQU KEY.F56	TO terminfo$<578>
EQU KEY.F57	TO terminfo$<579>
EQU KEY.F58	TO terminfo$<580>
EQU KEY.F59	TO terminfo$<581>
EQU KEY.F60	TO terminfo$<582>
EQU KEY.F61	TO terminfo$<583>
EQU KEY.F62	TO terminfo$<584>
EQU KEY.F63	TO terminfo$<585>
EQU CLEAR.MARGINS	TO terminfo$<586>
EQU SET.LEFT.MARGIN	TO terminfo$<587>
EQU SET.RIGHT.MARGIN	TO terminfo$<588>
EQU LABEL.KEY.FUNCTION.17	TO terminfo$<589>
EQU LABEL.KEY.FUNCTION.18	TO terminfo$<590>
EQU LABEL.KEY.FUNCTION.19	TO terminfo$<591>
EQU LABEL.KEY.FUNCTION.20	TO terminfo$<592>
EQU LABEL.KEY.FUNCTION.2	TO terminfo$<593>
EQU LABEL.KEY.FUNCTION.22	TO terminfo$<594>
EQU LABEL.KEY.FUNCTION.2	TO terminfo$<595>
EQU LABEL.KEY.FUNCTION.24	TO terminfo$<596>
EQU LABEL.KEY.FUNCTION.25	TO terminfo$<597>
EQU LABEL.KEY.FUNCTION.26	TO terminfo$<598>
EQU LABEL.KEY.FUNCTION.27	TO terminfo$<599>
EQU LABEL.KEY.FUNCTION.28	TO terminfo$<600>

TERMINFO EQUATEs (Continued)

EQU LABEL.KEY.FUNCTION.2	TO terminfo$<601>
EQU LABEL.KEY.FUNCTION.30	TO terminfo$<602>
EQU LABEL.KEY.FUNCTION.31	TO terminfo$<603>
EQU LABEL.KEY.FUNCTION.32	TO terminfo$<604>
EQU LABEL.KEY.FUNCTION.33	TO terminfo$<605>
EQU LABEL.KEY.FUNCTION.34	TO terminfo$<606>
EQU LABEL.KEY.FUNCTION.35	TO terminfo$<607>
EQU LABEL.KEY.FUNCTION.36	TO terminfo$<608>
EQU LABEL.KEY.FUNCTION.37	TO terminfo$<609>
EQU LABEL.KEY.FUNCTION.38	TO terminfo$<610>
EQU LABEL.KEY.FUNCTION.39	TO terminfo$<611>
EQU LABEL.KEY.FUNCTION.40	TO terminfo$<612>
EQU LABEL.KEY.FUNCTION.41	TO terminfo$<613>
EQU LABEL.KEY.FUNCTION.42	TO terminfo$<614>
EQU LABEL.KEY.FUNCTION.43	TO terminfo$<615>
EQU LABEL.KEY.FUNCTION.44	TO terminfo$<616>
EQU LABEL.KEY.FUNCTION.45	TO terminfo$<617>
EQU LABEL.KEY.FUNCTION.46	TO terminfo$<618>
EQU LABEL.KEY.FUNCTION.4	TO terminfo$<619>
EQU LABEL.KEY.FUNCTION.48	TO terminfo$<620>
EQU LABEL.KEY.FUNCTION.49	TO terminfo$<621>
EQU LABEL.KEY.FUNCTION.50S	TO terminfo$<622>
EQU LABEL.KEY.FUNCTION.51	TO terminfo$<623>
EQU LABEL.KEY.FUNCTION.52	TO terminfo$<624>
EQU LABEL.KEY.FUNCTION.53	TO terminfo$<625>
EQU LABEL.KEY.FUNCTION.54	TO terminfo$<626>
EQU LABEL.KEY.FUNCTION.55	TO terminfo$<627>
EQU LABEL.KEY.FUNCTION.56	TO terminfo$<628>
EQU LABEL.KEY.FUNCTION.57	TO terminfo$<629>
EQU LABEL.KEY.FUNCTION.58	TO terminfo$<630>
EQU LABEL.KEY.FUNCTION.59	TO terminfo$<631>
EQU LABEL.KEY.FUNCTION.60	TO terminfo$<632>
EQU LABEL.KEY.FUNCTION.61	TO terminfo$<633>
EQU LABEL.KEY.FUNCTION.62	TO terminfo$<634>
EQU LABEL.KEY.FUNCTION.63	TO terminfo$<635>

Example

```
$INCLUDE UNIVERSE.INCLUDE TERMINFO
PRINT AT.NEGATIVE.1
PRINT "Your terminal type is":TAB:TERMINAL.NAME
```

The program output on the cleared screen is:

```
Your terminal type is icl6404|ICL 6404CG Color Video Display
```

TIME function

Syntax

TIME ( )

Description

Use the TIME function to return a string value expressing the internal time of day. The internal time is the number of seconds that have passed since midnight to the nearest thousandth of a second (local time).

The parentheses must be used with the TIME function to distinguish it from a user-named variable called TIME. However, no arguments are required with the TIME function.

UNIX System V. The time is returned only to the nearest whole second.

If the TIME.MILLISECOND option of the $OPTIONS statement is set, the TIME function returns the system time in whole seconds.

Example

```
PRINT TIME()
```

This is the program output:

```
40663.842
```

Syntax

TIMEDATE ( )

Description

Use the TIMEDATE function to return the current system time and date in the following format:

hh:*mm*:*ss dd mmm yyyy*

hh	Hours (based on a 24-hour clock)
mm	Minutes
ss	Seconds
dd	Day
mmm	Month
yyyy	Year

The parentheses must be used with the TIMEDATE function to distinguish it from a user-named variable called TIMEDATE. However, no arguments are required with the TIMEDATE function.

If you want to increase the number of spaces between the time and the date, edit the line beginning with TMD0001 in the *msg.txt* file in the UV account directory. This line can contain up to four pound signs (#), also called hash signs. Each # prints a space between the time and the date.

Examples

```
PRINT TIMEDATE()
```

This is the program output:

```
11:19:07 18 JUN 1996
```

If the TMD0001 message contains four #s, the program output is:

```
11:19:07    18 JUN 1996
```

TIMEOUT statement

Syntax

TIMEOUT {*file.variable* | *link.number*}, *time*

Description

Use the TIMEOUT statement to terminate a READSEQ or READBLK statement if no data is read in the specified time. You can also use the TIMEOUT statement to set a time limit for a UV/Net link. Use the TTYGET and TTYSET statements to set a timeout value for a file open on a serial communications port.

The TIMEOUT statement is not supported on Windows NT.

file.variable specifies a file opened for sequential access.

time is an expression that evaluates to the number of seconds the program should wait before terminating the READSEQ or READBLK statement or the UV/Net connections.

link.number is the UV/Net link. It is a positive number from 1 through 255 (or the number set in the NET_MAXCONNECT VALUE for UV/Net connections).

TIMEOUT causes subsequent READSEQ and READBLK statement to terminate and execute their ELSE statements if the number of seconds specified by *time* elapses while waiting for data. Use the STATUS function to determine if *time* has elapsed. In the event of a timeout, neither READBLK nor READSEQ returns any bytes from the buffer, and the entire I/O operation must be retried.

If either *file.variable* or *time* evaluates to the null value, the TIMEOUT statement fails, the program terminates, and a run-time error message appears.

Examples

```
TIMEOUT SUN.MEMBER, 10
READBLK VAR1 FROM SUN.MEMBER, 15 THEN PRINT VAR1 ELSE
   IF STATUS() = 2 THEN
    PRINT "TIMEOUT OCCURRED"
   END ELSE
    PRINT "CANNOT OPEN FILE"
   END
   GOTO EXIT.PROG
END
```

This is the program output:

```
TIMEOUT OCCURRED
```

The following example sets a 30-second timeout for the UV/Net connection to the system ORION:

```
TIMEOUT SYSTEM (1200, "ORION"), 30
OPEN "ORION!/u1/user/file" TO FU.ORIONFILE
READ X,Y FROM FU.ORIONFILE
   ELSE
    IF SYSTEM (1203)= 81015
    THEN PRINT "TIMEOUT ON READ"
   END
   ELSE
    PRINT "READ ERROR"
   END
END
```

TPARM function

Syntax

TPARM (*terminfo.string*, [*arg1*], [*arg2*], [*arg3*], [*arg4*], [*arg5*], [*arg6*], [*arg7*], [*arg8*])

Description

Use the TPARM function to evaluate a parameterized terminfo string.

terminfo.string represents a string of characters to be compiled by the terminfo compiler, *tic*. These terminal descriptions define the sequences of characters to send to the terminal to perform special functions. *terminfo.string* evaluates to one of four types of capability: numeric, Boolean, string, or parameterized string. If *terminfo.string* or any of the eight arguments evaluates to the null value, the TPARM function fails, the program terminates, and a run-time error message appears.

Numeric capabilities are limited to a length of five characters that must form a valid number. Only nonnegative numbers 0 through 32,767 are allowed. If a value for a particular capability does not apply, the field should be left blank.

Boolean capabilities are limited to a length of one character. The letter Y (in either uppercase or lowercase) indicates that the specified capability is present. Any value other than Y indicates that the specified capability is not present.

String capabilities are limited to a length of 44 characters. You can enter special characters as follows:

\E or \e	The ESC character (ASCII 27).
\n or \l	The LINEFEED character (ASCII 10).
\r	The RETURN character (ASCII 13).
\t	The TAB character (ASCII 9).
\b	The BACKSPACE character (ASCII 8).
\f	The formfeed character (ASCII 12).
\s	A space (ASCII 32).
^*x*	The representation for a control character (ASCII 0 through 31). The character can be either uppercase or lowercase. A list of some control character representations follows:

Representation	Control Character
^A	^a
ASCII 1 (**Ctrl-A**)	ASCII 1 (**Ctrl-A**)
^@	ASCII 0
^[	ASCII 27 (**Esc**)
^\	ASCII 28
^]	ASCII 29
^^	ASCII 30
^_	ASCII 31
^?	ASCII 127 (**Del**)

nnn	Represents the ASCII character with a value of *nnn* in octal—for example \033 is the **Esc** character (ASCII 27).
\\	Represents the "\" character.
\,	Represents the "," character.
\^	Represents the "^" character.

Parameterized string capabilities, such as cursor addressing, use special encoding to include values in the appropriate format. The parameter mechanism is a stack with several commands to manipulate it:

%p*n*	Push parameter number *n* onto the stack. Parameters number 1 through 8 are allowed and are represented by *arg1* through *arg8* of the TPARM function.
%'*c*'	The ASCII value of character *c* is pushed onto the stack.
%[*nnn*]	Decimal number *nnn* is pushed onto the top of the stack.
%d	Pop the top parameter off the stack, and output it as a decimal number.
%*n*d	Pop the top parameter off the stack, and output it as a decimal number in a field *n* characters wide.
%0*n*d	Like %*n*d, except that 0s are used to fill out the field.
%c	The top of the stack is taken as a single ASCII character and output.
%s	The top of the stack is taken as a string and output.

%+ %– %* %/

The top two elements are popped off the stack and added, subtracted, multiplied, or divided. The result is pushed back on the stack. The fractional portion of a quotient is discarded.

%m The second element on the stack is taken modulo of the first element, and the result is pushed onto the stack.

%& %| %^ The top two elements are popped off the stack and a bitwise AND, OR, or XOR operation is performed. The result is pushed onto the stack.

%= %< %> The second element on the stack is tested for being equal to, less then, or greater than the first element. If the comparison is true, a 1 is pushed onto the stack, otherwise a 0 is pushed.

%! %~ The stack is popped, and either the logical or the bitwise NOT of the first element is pushed onto the stack.

%i One (1) is added to the first two parameters. This is useful for terminals that use a one-based cursor address, rather than a zero-based.

%P*x* Pop the stack, and put the result into variable *x*, where *x* is a lowercase letter (a – z).

%g*x* Push the value of variable *x* on the top of the stack.

%? *exp* %t *exp* [%e *exp*] %;

Form an if-then-else expression, with "%?" representing "IF", "%t" representing "THEN", "%e" representing "ELSE", and "%;" terminating the expression. The else expression is optional. Else-If expressions are possible. For example:

```
%? C1 %t B1 %e C2 %t B2 %e C3 %t B3 %e C4 %t B4 %e %
```

C*n* are conditions, and B*n* are bodies.

%% Output a percent sign (%).

A delay in milliseconds can appear anywhere in a string capability. A delay is specified by $<*nnn*>, where *nnn* is a decimal number indicating the number of milliseconds (one thousandth of a second) of delay desired. A proper number of delay characters will be output, depending on the current baud rate.

Syntax

TPRINT [ON *print.channel*] [*print.list*]

Description

Use the TPRINT statement to send data to the screen, a line printer, or another print file. TPRINT is similar to the PRINT statement, except that TPRINT lets you specify time delay expressions in the print list.

The ON clause specifies the logical print channel to use for output. *print.channel* is an expression that evaluates to a number from –1 through 255. If you do not use the ON clause, logical print channel 0 is used, which prints to the user's terminal if PRINTER OFF is set (see the PRINTER statement). If *print.channel* evaluates to the null value, the TPRINT statement fails, the program terminates, and a run-time error message appears. Logical print channel –1 prints the data on the screen, regardless of whether a PRINTER ON statement has been executed.

You can specify HEADING, FOOTING, PAGE, and PRINTER CLOSE statements for each logical print channel. The contents of the print files are printed in order by logical print channel number.

print.list can contain any BASIC expression. The elements of the list can be numeric or character strings, variables, constants, or literal strings. The list can consist of a single expression or a series of expressions separated by commas (,) or colons (:) for output formatting. If no *print.list* is designated, a blank line is printed. The null value cannot be printed.

print.list can also contain time delays of the form $<*time*>. *time* is specified in milliseconds to the tenth of a millisecond. As the print list is processed, each time delay is executed as it is encountered.

Expressions separated by commas are printed at preset tab positions. The default tabstop setting is 10 characters. See the TABSTOP statement for information about changing the default setting. Use multiple commas together for multiple tabulations between expressions.

Expressions separated by colons are concatenated. That is, the expression following the colon is printed immediately after the expression preceding the colon. To print a list without a LINEFEED and RETURN, end *print.list* with a colon (:).

Example

The following example prints the string ALPHA followed by a delay of 1 second, then the letters in the variable X. The printing of each letter is followed by a delay of one tenth of a second.

```
X="A$<100>B$<100>C$<100>D$<100>E"
TPRINT "ALPHA$<1000.1> ":X
```

This is the program output:

```
ALPHA ABCDE
```

Syntax

TRANS ([DICT] *filename, record.ID, field#, control.code*)

Description

Use the TRANS function to return the contents of a field or a record in a UniVerse file. TRANS opens the file, reads the record, and extracts the specified data.

filename is an expression that evaluates to the name of the remote file. If TRANS cannot open the file, a run-time error occurs, and TRANS returns an empty string.

record.ID is an expression that evaluates to the ID of the record to be accessed.

field# is an expression that evaluates to the number of the field from which the data is to be extracted. If *field#* is –1, the entire record is returned, except for the record ID.

control.code is an expression that evaluates to a code specifying what action to take if data is not found or is the null value. The possible control codes are:

X (default) Returns an empty string if the record does not exist or data cannot be found.

V Returns an empty string and produces an error message if the record does not exist or data cannot be found.

C Returns the value of *record.ID* if the record does not exist or data cannot be found.

N Returns the value of *record.ID* if the null value is found.

Note that the returned value is lowered. For example, value marks in the original field become subvalue marks in the returned value. For more information, see the LOWER function in Volume 1.

If *record.ID* is multivalued, the translation occurs for each record ID and the result is multivalued (value marks separate data translated from each record).

If *filename*, *record.ID*, or *field#* evaluates to the null value, the TRANS function fails, the program terminates, and a run-time error message appears. If *control.code* evaluates to the null value, null is ignored and X is used.

The TRANS function is the same as the XLATE function.

Example

```
X=TRANS("VOC","EX.BASIC",1,"X")
PRINT "X= ":X
*
FIRST=TRANS("SUN.MEMBER","6100",2,"X")

LAST=TRANS("SUN.MEMBER","6100",1,"X")
PRINT "NAME IS ":FIRST:" ":LAST
```

This is the program output:

```
X= F BASIC examples file
NAME IS BOB MASTERS
```

Syntax

BEGIN TRANSACTION
[*statements*]

{ COMMIT [WORK] | ROLLBACK [WORK] }
[*statements*]

[{ COMMIT [WORK] | ROLLBACK [WORK] }
[*statements*]
.
.
.]

END TRANSACTION

PI/open Syntax

TRANSACTION START
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

TRANSACTION COMMIT
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

TRANSACTION ABORT

Description

Use transaction statements to treat a sequence of file I/O operations as one logical operation with respect to recovery and visibility to other users. These operations can include file I/O operations or subtransactions. BASIC accepts PI/open syntax in addition to UniVerse syntax.

Note: You cannot mix both types of syntax within a program.

For more information about transaction statements, refer to *UniVerse BASIC Programmer's Guide*.

TRANSACTION ABORT statement

Syntax

TRANSACTION ABORT

Description

Use the TRANSACTION ABORT statement to cancel all file I/O changes made during a transaction.

You can use the TRANSACTION ABORT statement in a transaction without a TRANSACTION COMMIT statement to review the results of a possible change. Doing so does not affect the parent transaction or the database.

After the transaction ends, execution continues with the statement following the TRANSACTION ABORT statement.

Example

The following example shows the use of the TRANSACTION ABORT statement to terminate a transaction if both the ACCOUNTS RECEIVABLE file and the INVENTORY file cannot be successfully updated:

```
PROMPT ''
OPEN 'ACC.RECV' TO ACC.RECV ELSE STOP 'NO OPEN ACC.RECV'
OPEN 'INVENTORY' TO INVENTORY ELSE STOP 'NO OPEN INVENTORY'

PRINT 'Customer Id : ':
INPUT CUST.ID
PRINT 'Item No.  : ':
INPUT ITEM
PRINT 'Amount   : ':
INPUT AMOUNT

* Start a transaction to ensure both or neither records
* updated
TRANSACTION START ELSE STOP 'Transaction start failed.'
* Read customer record from accounts receivable
  READU ACT.REC FROM ACC.RECV, CUST.ID
  ON ERROR
   STOP 'Error reading ':CUST.ID:' from ACC.RECV file.'
  END LOCKED
   * Could not lock record so ABORT transaction
   TRANSACTION ABORT
   STOP 'Record ':CUST.ID:' on file ACC.RECV locked by user
```

```
':STATUS()
  END THEN
   * Build new record
   ACT.REC<1,-1> = ITEM:@SM:AMOUNT
   ACT.REC<2> = ACT.REC<2> + AMOUNT
  END ELSE
   * Create new record
   ACT.REC = ITEM:@SM:AMOUNT:@FM:AMOUNT
  END
  * Read item record from inventory
  READU INV.REC FROM INVENTORY, ITEM
  ON ERROR
   STOP 'Error reading ':ITEM:' from INVENTORY file.'
  END LOCKED
   * Could not lock record so ABORT transaction
   TRANSACTION ABORT
   STOP 'Record ':ITEM:' on file INVENTORY locked by user
':STATUS()
  END THEN
   * Build new record
   INV.REC<1> = INV.REC<1> - 1
   INV.REC<2> = INV.REC<2> - AMOUNT
  END ELSE
   STOP 'Record ':ITEM:' is not on file INVENTORY.'
  END
  * Write updated records to accounts receivable and inventory
  WRITEU ACT.REC TO ACC.RECV, CUST.ID
  WRITEU INV.REC TO INVENTORY, ITEM

TRANSACTION COMMIT ELSE STOP 'Transaction commit failed.'

END
```

TRANSACTION COMMIT statement

Syntax

TRANSACTION COMMIT
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the TRANSACTION COMMIT statement to commit all file I/O changes made during a transaction.

The TRANSACTION COMMIT statement can either succeed or fail. If the TRANSACTION COMMIT statement succeeds, the THEN statements are executed; any ELSE statements are ignored. If the TRANSACTION COMMIT statement fails, the ELSE statements, if present, are executed, and control is transferred to the statement following the TRANSACTION COMMIT statement.

Syntax

```
TRANSACTION START
        {THEN statements [ELSE statements] | ELSE statements}
```

Description

Use the TRANSACTION START statement to begin a new transaction.

THEN and ELSE Clauses

You must have a THEN clause or an ELSE clause, or both, in a TRANSACTION START statement.

If the TRANSACTION START statement successfully begins a transaction, the statements in the THEN clause are executed. If for some reason UniVerse is unable to start the transaction, a fatal error occurs, and you are returned to the UniVerse system prompt.

TRIM function

Syntax

TRIM (*expression* [,*character* [,*option*]])

Description

Use the TRIM function to remove unwanted characters in *expression*. If only *expression* is specified, multiple occurrences of spaces and tabs are reduced to a single tab or space, and all leading and trailing spaces and tabs are removed.

character specifies a character other than a space or a tab. If only *expression* and *character* are specified, multiple occurrences of *character* are replaced with a single occurrence, and leading and trailing occurrences of *character* are removed.

option specifies the type of trim operation to be performed:

A Remove all occurrences of *character*

B Remove both leading and trailing occurrences of *character*

D Remove leading, trailing, and redundant spaces and tabs

E Remove trailing spaces and tabs

F Remove leading spaces and tabs

L Remove all leading occurrences of *character*

R Remove leading, trailing, and redundant occurrences of *character*

T Remove all trailing occurrences of *character*

If *expression* evaluates to the null value, null is returned. If *option* evaluates to the null value, null is ignored and option R is assumed. If *character* evaluates to the null value, the TRIM function fails, the program terminates, and a run-time error message appears.

Example

```
A="  Now is the time  for  all good men to"
PRINT A
PRINT TRIM(A)
```

This is the program output:

```
  Now is the time  for  all good men to
Now is the time for all good men to
```

Syntax

TRIMB (*expression*)

Description

Use the TRIMB function to remove all trailing spaces and tabs from *expression*. All other spaces or tabs in *expression* are left intact. If *expression* evaluates to the null value, null is returned.

Example

```
A="  THIS IS A  SAMPLE STRING  "
PRINT "'":A:"'": " IS THE STRING"
PRINT "'":TRIMB(A):"'":" IS WHAT TRIMB DOES"
END
```

This is the program output:

```
'  THIS IS A  SAMPLE STRING  ' IS THE STRING
'  THIS IS A  SAMPLE STRING' IS WHAT TRIMB DOES
```

TRIMBS function

Syntax

TRIMBS (*dynamic.array*)

CALL –TRIMBS (*return.array*, *dynamic.array*)

Description

Use the TRIMBS function to remove all trailing spaces and tabs from each element of *dynamic.array.*

TRIMBS removes all trailing spaces and tabs from each element and reduces multiple occurrences of spaces and tabs to a single space or tab.

If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is null, null is returned for that value.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

Syntax

TRIMF (*expression*)

Description

Use the TRIMF function to remove all leading spaces and tabs from *expression*. All other spaces or tabs in *expression* are left intact. If *expression* evaluates to the null value, null is returned.

Example

```
A="  THIS IS A  SAMPLE STRING  "
PRINT "'":A:"'":" IS THE STRING"
PRINT "'":TRIMF(A):"'":" IS WHAT TRIMF DOES"
END
```

This is the program output:

```
'  THIS IS A  SAMPLE STRING  ' IS THE STRING
'THIS IS A  SAMPLE STRING  ' IS WHAT TRIMF DOES
```

TRIMFS function

Syntax

TRIMFS (*dynamic.array*)

CALL –TRIMFS (*return.array, dynamic.array*)

Description

Use the TRIMFS function to remove all leading spaces and tabs from each element of *dynamic.array.*

TRIMFS removes all leading spaces and tabs from each element and reduces multiple occurrences of spaces and tabs to a single space or tab.

If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is null, null is returned for that value.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

Syntax

TRIMS (*dynamic.array*)

CALL –TRIMS (*return.array, dynamic.array*)

Description

Use the TRIMS function to remove unwanted spaces and tabs from each element of *dynamic.array.*

TRIMS removes all leading and trailing spaces and tabs from each element and reduces multiple occurrences of spaces and tabs to a single space or tab.

If *dynamic.array* evaluates to the null value, null is returned. If any element of *dynamic.array* is null, null is returned for that value.

If you use the subroutine syntax, the resulting dynamic array is returned as *return.array.*

TTYCTL statement

Syntax

TTYCTL *file.variable*, *code#*
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the TTYCTL statement to set terminal device characteristics on Berkeley terminal drivers. *code#* specifies the action to take.

This statement is not supported on UNIX System V or Windows NT.

The following table lists the available actions:

TTYCTL Action Codes

Argument	Action
0	No operation, determines if a device is a TTY.
1	Sets HUP (hang up data line) on close of file.
2	Clears HUP on close of file.
3	Sets exclusive use flag for TTY.
4	Resets exclusive use flag.
5	Sets the BREAK.
6	Clears the BREAK.
7	Turns on DTR (Data Terminal Ready).
8	Turns off DTR.
9	Flushes input and output buffers.
10	Waits for the output buffer to drain.

file.variable specifies a file previously opened for sequential access to a terminal device. If *file.variable* evaluates to the null value, the TTYCTL statement fails, the program terminates, and a run-time error message appears.

If the action is taken, the THEN statements are executed. If no THEN statements are present, program execution continues with the next statement.

If an error is encountered during the execution of the TTYCTL operation, or if the file variable is not open to a terminal device, the ELSE statements are executed; any THEN statements are ignored.

Example

```
OPENSEQ 'FILE.E', 'RECORD4' TO FILE ELSE ABORT
*
TTYCTL FILE, 0
   THEN PRINT 'THE FILE IS A TTY'
   ELSE PRINT 'THE FILE IS NOT A TTY'
```

This is the program output:

```
THE FILE IS NOT A TTY
```

TTYGET statement

Syntax

TTYGET *variable* [FROM {*file.variable* | LPTR [*n*] | MTU [*n*] }]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the TTYGET statement to assign the characteristics of a terminal, line printer channel, or tape unit as a dynamic array to *variable*. If the FROM clause is omitted, a dynamic array of the terminal characteristics for your terminal is assigned to *variable*.

file.variable is a terminal opened for sequential processing with the OPENDEV or OPENSEQ statement. If *file.variable* is specified, the terminal characteristics for the specified terminal are retrieved.

n specifies a logical print channel with LPTR or a tape unit with MTU. (You cannot specify a tape unit on Windows NT.) If *n* is specified, the characteristics for the print channel or tape unit are retrieved. For logical print channels *n* is in the range of 0 through 225; the default is 0. For tape units *n* is in the range of 0 through 7; the default is 0.

If the terminal characteristics are retrieved, the THEN statements are executed.

If the device does not exist or cannot be opened, or if no dynamic array is returned, the ELSE statements are executed; any THEN statements are ignored.

If either *file.variable* or *n* evaluates to the null value, the TTYGET statement fails, the program terminates, and a run-time error message appears.

The best way to access the information in the dynamic array is to include the BASIC code UNIVERSE.INCLUDE TTY. The syntax for including this file is:

```
$INCLUDE UNIVERSE.INCLUDE TTY
```

This file equates each value of the dynamic array to a name, so that each value can be easily accessed in your program. To take advantage of this code you must call variable *tty$*. Once this code has been included in your program, you can use the names to access the values of the dynamic array. To set values for a terminal line, use the TTYSET statement.

The following table lists the equate names to the values of the dynamic array, and describes each value. The final columns indicate which values are available on

different operating systems: SV indicates System V, B indicates Berkeley UNIX, and NT indicates Windows NT.

TTYGET Statement Values

Value	Name	Description	Availability		
			SV	B	NT
Field 1					
1	MODE.TYPE	One of these modes:			
		MODE$LINE or 0 = line	✓	✓	
		MODE$RAW or 1 = raw	✓	✓	✓
		MODE$CHAR or 2 = character	✓	✓	
		MODE$EMULATE or 3 = emulated	✓	✓	✓
2	MODE.MIN	Minimum number of characters before input.	✓	✓	✓
3	MODE.TIME	Minimum time in milliseconds before input.	✓	✓	✓
Field 2					
1	CC.INTR	Interrupt character. –1 undefined.	✓	✓	✓
2	CC.QUIT	Quit character. –1 undefined.	✓	✓	
3	CC.SUSP	Suspend character. –1 undefined.	✓	✓	
4	CC.DSUSP	dsusp character. –1 undefined.		✓	
5	CC.SWITCH	Switch character. –1 undefined.	✓		
6	CC.ERASE	erase character. –1 undefined.	✓	✓	✓
7	CC.WERASE	werase character. –1 undefined.		✓	
8	CC.KILL	Kill character. –1 undefined.	✓	✓	✓
9	CC.LNEXT	lnext character. –1 undefined.		✓	
10	CC.RPRINT	rprint character. –1 undefined.		✓	✓
11	CC.EOF	eof character. –1 undefined.	✓	✓	
12	CC.EOL	eol character. –1 undefined.	✓	✓	
13	CC.EOL2	eol2 character. –1 undefined.	✓		
14	CC.FLUSH	Flush character. –1 undefined.		✓	

TTYGET Statement Values (Continued)

Value	Name	Description	Availability		
			SV	B	NT
15	CC.START	Start character. –1 undefined. On System V, ^Q only.	✓	✓	✓
16	CC.STOP	Stop character. –1 undefined. On System V, ^S only.	✓	✓	✓
17	CC.LCONT	lcont character. –1 undefined. Emulated only.	✓	✓	✓
18	CC.FMC	*fmc* character. –1 undefined. Emulated only.	✓	✓	✓
19	CC.VMC	*vmc* character. –1 undefined. Emulated only.	✓	✓	✓
20	CC.SMC	*smc* character. –1 undefined. Emulated only.	✓	✓	✓
21	CCDEL	Delete character.	✓	✓	
Field 3					
1	CARRIER.RECEIVE	Terminal can receive data.	✓	✓	✓
2	CARRIER.HANGUP	Hang up upon close of terminal.	✓	✓	
3	CARRIER.LOCAL	Terminal is a local line.	✓	✓	✓
Field 4					
1	CASE.UCIN	Convert lowercase to uppercase on input.	✓	✓	
2	CASE.UCOUT	Convert lowercase to uppercase on output.	✓	✓	
3	CASE.XCASE	Uppercase is preceded by a backslash (\ ) to distinguish it from lowercase.	✓	✓	
4	CASE.INVERT	Invert case on input. Emulated only.	✓	✓	✓
Field 5					
1	CRMODE.INLCR	Convert LINEFEED to RETURN on input.	✓	✓	
2	CRMODE.IGNCR	Ignore RETURN on input.	✓	✓	
3	CRMODE.ICRNL	Convert RETURN to LINEFEED on input.	✓	✓	
4	CRMODE.ONLCR	Convert LINEFEED to LINEFEED, RETURN on output.	✓	✓	
5	CRMODE.OCRNL	Convert RETURN to LINEFEED on output.	✓	✓	

TTYGET Statement Values (Continued)

Value	Name	Description	Availability SV	Availability B	Availability NT
6	CRMODE.ONOCR	Prohibit output of RETURN when cursor is in column 0.	✓	✓	
7	CRMODE.ONLRET	LINEFEED performs RETURN function.	✓	✓	
Field 6					
1	DELAY.BS	Set backspace delay.	✓	✓	
2	DELAY.CR	Set RETURN delay.	✓	✓	
3	DELAY.FF	Set formfeed delay.	✓	✓	
4	DELAY.LF	Set LINEFEED delay.	✓	✓	
5	DELAY.VT	Set vertical tab delay.	✓	✓	
6	DELAY.TAB	Set tab delay.	✓	✓	
7	DELAY.FILL	0 = time delay 1 = fill with empty strings 2 = fill with DELETEs	✓	✓	
Field 7					
1	ECHO.ON	Set terminal echo on.	✓	✓	✓
2	ECHO.ERASE	ECHOE$ERASE or 0 = print echo character ECHOE$BS or 1 = echo as backspace ECHOE$BSB or 2 = echo as backspace, space, backspace ECHOE$PRINTER or 3 = echo as a printer	✓	✓	
3	ECHO.KILL	ECHOK$KILL or 0 = kill as kill character ECHOK$LF or 1 = kill as RETURN, LINEFEED ECHOK$ERASE or 2 = kill as series of erases	✓	✓	
4	ECHO.CTRL	Set control to echo as ^ character	✓	✓	
5	ECHO.LF	When echo is off, echo RETURN as RETURN, LINEFEED	✓	✓	✓

TTYGET Statement Values (Continued)

Value	Name	Description	Avail-ability		
			SV	B	NT
Field 8					
1	HANDSHAKE.XON	1 = turns on X-ON / X-OFF protocol 0 = turns off X-ON / X-OFF protocol	✓	✓	✓
2	HANDSHAKE.STARTANY	1 = any characters acts as X-ON 0 = only X-ON character acts as X-ON	✓	✓	
3	HANDSHAKE.TANDEM	1 = when input buffer is nearly full, X-OFF is sent 0 = turns off automatic X-OFF, X-ON mode	✓	✓	✓
4	HANDSHAKE.DTR	1 = turns on DTR 0 = turns off DTR	✓	✓	
Field 9					
1	OUTPUT.POST	Output postprocessing occurs.	✓	✓	
2	OUTPUT.TILDE	Special output processing for tilde.	✓	✓	
3	OUTPUT.BG	Stop background processes at output.	✓	✓	
4	OUTPUT.CS	Output clearscreen before reports. Emulated only.	✓	✓	
5	OUTPUT.TAB	Set output tab expansion.	✓	✓	
Field 10					
1	PROTOCOL.LINE	Line protocol	✓	✓	
2	PROTOCOL.BAUD	1 = 50 9 = 1200 2 = 75 10 = 1800 3 = 110 11 = 2400 4 = 134 12 = 4800 5 = 150 13 = 9600 6 = 200 14 or EXTA = 19200 7 = 300 15 = EXTB 8 = 600	✓	✓	✓

TTYGET Statement Values (Continued)

Value	Name	Description	Availability SV	Availability B	Availability NT
3	PROTOCOL.DATA	Character size: 5 = 5 bits 7 = 7 bits 6 = 6 bits 8 = 8 bits	✓	✓	✓
4	PROTOCOL.STOP	2 = 2 stopbits 1 = 1 stopbit	✓	✓	✓
5	PROTOCOL.OUTPUT	Output parity:			
		0 = no parity			✓
		1 = even parity	✓	✓	✓
		2 = odd parity	✓	✓	✓
6	PROTOCOL.INPUT	Input parity:			
		0 = disable input parity checking	✓	✓	✓
		1 = enable input parity checking	✓	✓	✓
		2 = mark parity errors	✓	✓	✓
		3 = mark parity errors with a null	✓	✓	✓
		4 = ignore parity errors	✓	✓	
7	PROTOCOL.STRIP	1 = strip to 7 bits 0 = 8 bits	✓	✓	
Field 11					
1	SIGNALS.ENABLE	Enable signal keys: Interrupt, Suspend, Quit.	✓	✓	
2	SIGNALS.FLUSH	Flush type-ahead buffer.	✓	✓	
3	SIGNALS.BRKKEY	0 = break ignored 1 = break as interrupt 2 = break as null	✓	✓	

TTYSET statement

Syntax

TTYSET *dynamic.array* [ON {*file.variable* | LPTR [*n*] | MTU [*n*] }]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the TTYSET statement to set the characteristics of a terminal, line printer channel, or tape unit. If only *dynamic.array* is specified, the terminal characteristics for your terminal are set based on the contents of *dynamic.array. dynamic.array* is a dynamic array of eleven fields, each of which has multiple values. A description of the expected contents of each value of *dynamic.array* is given in the TTYGET statement.

file.variable is a terminal opened for sequential processing with the OPENDEV or OPENSEQ statement. If *file.variable* is specified, the terminal characteristics for the specified terminal are set.

n specifies a logical print channel with LPTR or a tape unit with MTU. If *n* is specified, the characteristics for the print channel or tape unit are set. *n* is in the range of 0 through 225 for logical print channels; the default is 0. *n* is in the range of 0 through 7 for tape units; the default is 0. **Windows NT.** You cannot specify a tape unit.

If the terminal characteristics are set, the THEN statements are executed.

If the device does not exist or cannot be opened, or if no dynamic array is returned, the ELSE statements are executed; any THEN statements are ignored.

If *dynamic.array, file.variable,* or *n* evaluates to the null value, the TTYSET statement fails, the program terminates, and a run-time error message appears.

To build *dynamic.array,* get the current values of the terminal line using the TTYGET statement, manipulate the values, and reset them with the TTYSET statement. The best way to access the information in the dynamic array is to include the BASIC code UNIVERSE.INCLUDE TTY. The syntax for including this file is:

```
$INCLUDE UNIVERSE.INCLUDE TTY
```

This file equates each value of *variable* from the TTYGET statement with a name, so that each value can be easily accessed in your program. To take advantage of this code you must call variable *tty$*. Once this code is included in your program, you can use the names to access the values of the dynamic array. The TTYGET Values

table lists the names equated to the values of the dynamic array and describes the values.

Timeout Handling

You can set the MODE.MIN and MODE.TIME values to define timeouts for read operations over a communications line. MODE.MIN specifies the minimum number of characters to be received. MODE.TIME specifies time in tenths of a second. The two values interact to provide four cases that can be used as follows.

Intercharacter Timer. When you set the values of both MODE.MIN and MODE.TIME to greater than 0, MODE.TIME specifies the maximum time interval allowed between successive characters received by the communication line in tenths of a second. Timing begins only after the first character is received.

Blocking Read. When you set the value of MODE.MIN to greater than 0 and MODE.TIME to 0, no time limit is set, so the read operation waits until the specified number of characters have been received (or a newline in the case of READSEQ).

Read Timer. When you set the value of MODE.MIN to 0 and MODE.TIME to greater than 0, MODE.TIME specifies how long the read operation waits for a character to arrive before timing out. If no characters are received in the time specified, the READSEQ and READBLK statements use the ELSE clause if there is one. If you use the NOBUF statement to turn off buffering, the timer is reset after each character is received.

Nonblocking Read. When you set the values of both MODE.MIN and MODE.TIME to 0, data is read as it becomes available. The read operation returns immediately.

- If any characters are received:
  - READBLK returns as many characters as specified in the *blocksize* argument, or all the characters received, whichever is fewer.
  - READSEQ returns characters up to the first newline, or all the characters received if no newline is received.
- If no characters are received, READSEQ and READBLK use the ELSE clause if there is one.

UNASSIGNED function

Syntax

UNASSIGNED (*variable*)

Description

Use the UNASSIGNED function to determine if *variable* is unassigned. UNASSIGNED returns 1 (true) if *variable* is unassigned. It returns 0 (false) if *variable* is assigned a value, including the null value.

Example

```
A = "15 STATE STREET"
C = 23
X = UNASSIGNED(A)
Y = UNASSIGNED(B)
Z = UNASSIGNED(C)
PRINT X,Y,Z
```

This is the program output:

```
0  1  0
```

Syntax

UNLOCK [*expression*]

Description

Use the UNLOCK statement to release a process lock set by the LOCK statement.

expression specifies an integer from 0 through 63. If *expression* is not specified, all locks are released (see the LOCK statement in Volume 1).

If *expression* evaluates to an integer outside the range of 0 through 63, an error message appears and no action is taken.

If *expression* evaluates to the null value, the UNLOCK statement fails, the program terminates, and a run-time error message appears.

Examples

The following example unlocks execution lock 60:

```
UNLOCK 60
```

The next example unlocks all locks set during the current login session:

```
UNLOCK
```

The next example unlocks lock 50:

```
X=10
UNLOCK 60-X
```

UPCASE function

Syntax

UPCASE (*expression*)

Description

Use the UPCASE function to change all lowercase letters in *expression* to uppercase. If *expression* evaluates to the null value, null is returned.

Example

```
A="This is an example of the UPCASE function: "
PRINT A
PRINT UPCASE(A)
```

This is the program output:

```
This is an example of the UPCASE function:
THIS IS AN EXAMPLE OF THE UPCASE FUNCTION:
```

Syntax

WEOF [UNIT (*mtu*)] {THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the WEOF statement to write an end-of-file (EOF) mark to tape.

The UNIT clause specifies the number of the tape drive unit. Tape unit 0 is used if no unit is specified.

mtu is an expression that evaluates to a three-digit code (decimal). Although the *mtu* expression is a function of the UNIT clause, the WEOF statement uses only the third digit (the *u*). Its value must be in the range of 0 through 7 (see the READT statement for details on the *mtu* expression). If *mtu* evaluates to the null value, the WEOF statement fails, the program terminates, and a run-time error message appears.

Before a WEOF statement is executed, a tape drive unit must be attached (assigned) to the user. Use the ASSIGN command to assign a tape unit to a user. If no tape unit is attached or if the unit specification is incorrect, the ELSE statements are executed.

Use the STATUS function after a WEOF statement is executed to determine the result of the operation. The value returned by the STATUS function for the WEOF statement is hardware-dependent, that is, it varies according to the characteristics of the specific tape drive. Consult the documentation for your tape drive for information about interpreting the values returned by the STATUS function.

Example

```
WEOF UNIT(007) ELSE PRINT "OPERATION NOT COMPLETED."
```

WEOFSEQ statement

Syntax

WEOFSEQ *file.variable* [ON ERROR *statements*]

Description

Use the WEOFSEQ statement to write an end-of-file (EOF) mark in a file opened for sequential access. The end-of-file mark is written at the current position and has the effect of truncating the file at this point. Any subsequent READSEQ statement has its ELSE statements executed.

file.variable specifies a file opened for sequential access. If *file.variable* evaluates to the null value, the WEOFSEQ statement fails, the program terminates, and a run-time error message appears.

Note: On Windows NT systems, you cannot use the WEOFSEQ statement with a diskette drive that you opened with the OPENDEV statement. For 1/4-inch cartridge tape drives (60 MB or 150 MB) you can use WEOFSEQ to write an end-of-file (EOF) mark at the beginning of the data or after a write.

The ON ERROR Clause

The ON ERROR clause is optional in the WEOFSEQ statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the WEOFSEQ statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

See the OPENSEQ, READSEQ, and WRITESEQ statements for more information about sequential file processing.

Note: Some systems do not support the truncation of disk files. WEOFSEQ is ignored on these systems, except that WEOFSEQ always works at the beginning of a file.

Example

The following example writes an end-of-file mark on the record RECORD in the file TYPE1:

```
OPENSEQ 'TYPE1','RECORD' TO FILE ELSE STOP
WEOFSEQ FILE
```

WRITE statements

Syntax

WRITE[U] *expression* {ON | TO} [*file.variable,*] *record.ID*
[ON ERROR *statements*] [LOCKED *statements*]
[THEN *statements*] [ELSE *statements*]

WRITEV[U] *expression* {ON | TO} [*file.variable,*] *record.ID, field#*
[ON ERROR *statements*] [LOCKED *statements*]
[THEN *statements*] [ELSE *statements*]

Description

Use WRITE statements to write new data to a record in a UniVerse file. The value of *expression* replaces any data previously stored in the record.

Effects of WRITE Statements

Use this statement…	To do this…
WRITE	Write to a record.
WRITEU	Write to a record, retaining an update record lock.
WRITEV	Write to a field.
WRITEVU	Write to a field, retaining an update record lock.

file.variable specifies an open file. If *file.variable* is not specified, the default file is assumed (for more information on default files, see the OPEN statement. If the file is neither accessible nor open, the program terminates and a run-time error message appears, unless ELSE statements are specified.

If, within a transaction, you try to write to a remote file over UV/Net, the write statement fails, the transaction is rolled back, the program terminates, and a run-time error message appears.

If the file is an SQL table, the effective user of the program must have SQL INSERT and UPDATE privileges to write records in the file. See the AUTHORIZATION statement in Volume 1 for information about the effective user of a program.

If the OPENCHK configurable parameter is set to TRUE, or if the file is opened with the OPENCHECK statement, all SQL integrity constraints are checked for every write to an SQL table. If an integrity check fails, the WRITE statement uses

the ELSE clause. Use the ICHECK function to determine what specific integrity constraint caused the failure.

The system searches the file for the record specified by *record.ID*. If the record is not found, WRITE creates a new record.

When updating a file, the WRITE statement releases the update record lock set with a READU statement. To maintain the update record lock set by the READU statement, use a WRITEU statement instead of a WRITE statement.

The WRITE statement does not strip trailing field marks enclosing empty strings from *expression*. Use the MATWRITE statement if that operation is required.

The new value is written to the record, and the THEN statements are executed. If no THEN statements are specified, execution continues with the statement following the WRITE statement. If WRITE fails, the ELSE statements are executed; any THEN statements are ignored.

If *expression* evaluates to the null value, the WRITE statement fails, the program terminates, and a run-time error message appears.

If *file.variable*, *record.ID*, or *field#* evaluates to the null value, all WRITE statements (that is, WRITE, WRITEU, WRITEV, WRITEVU) fail, the program terminates, and a run-time error message appears.

Use the STATUS function after a WRITE statement is executed, to determine the result of the operation, as follows:

0 The record was locked before the WRITE operation.

–2 The record was unlocked before the WRITE operation.

–3 The record failed an SQL integrity check.

The ON ERROR Clause

The ON ERROR clause is optional in WRITE statements. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered during processing of the WRITE statement.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.

- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

The LOCKED Clause

The LOCKED clause is optional, but recommended. Its format is the same as that of the ELSE clause.

The LOCKED clause handles a condition caused by a conflicting lock (set by another user) that prevents the WRITE statement from processing. The LOCKED clause is executed if one of the following conflicting locks exists:

- Exclusive file
- Intent file
- Shared file
- Update record
- Shared record

If the WRITE statement does not include a LOCKED clause, and a conflicting lock exists, the program pauses until the lock is released.

If a LOCKED clause is used, the value returned by the STATUS function is the terminal number of the user who owns the conflicting lock.

The WRITEU Statement

Use the WRITEU statement to update a record without releasing the update record lock set by a previous READU statement (see the READ statement). To release the update record lock set by a READU statement and maintained by a WRITEU statement, you must use a RELEASE, WRITE, MATWRITE, or WRITEV statement. If you do not explicitly release the lock, the record remains locked until the program executes the STOP statement. When more than one program or user could modify the same record, use a READU statement to lock the record before doing the WRITE or WRITEU.

If *expression* evaluates to the null value, the WRITEU statement fails, the program terminates, and a run-time error message appears.

The WRITEV Statement

Use the WRITEV statement to write a new value to a specified field in a record. The WRITEV statement requires that *field#* be specified. *field#* is the number of the field to which *expression* is written. It must be greater than 0. If either the record or the field does not exist, WRITEV creates them.

If *expression* evaluates to the null value, null is written to *field#*, provided that the field allows nulls. If the file is an SQL table, existing SQL security and integrity constraints must allow the write.

The WRITEVU Statement

Use the WRITEVU statement to update a specified field in a record without releasing the update record lock set by a previous READU statement (see the READ statement). The WRITEVU syntax is like that of the WRITEV and WRITEU statements.

If *expression* evaluates to the null value, null is written to *field#*, provided that the field allows nulls. If the file is an SQL table, existing SQL security and integrity constraints must allow the write.

Example

```
CLEAR
DATA "ELLEN","KRANZER","3 AMES STREET","CAMBRIDGE"
DATA "MA","02139","SAILING"
OPEN '','SUN.MEMBER' TO FILE ELSE
   PRINT "COULD NOT OPEN FILE"
   STOP
END
PRINT "ENTER YOUR FIRST NAME"
INPUT FNAME
PRINT "ENTER YOUR LAST NAME"
INPUT LNAME
PRINT "ENTER YOUR ADDRESS (PLEASE WAIT FOR PROMPTS)"
PRINT "STREET ADDRESS"
INPUT STREET
PRINT "ENTER CITY"
INPUT CITY
```

```
PRINT "ENTER STATE"
INPUT STATE
PRINT "ENTER ZIP CODE"
INPUT ZIP
PRINT "ENTER YOUR INTERESTS"
INPUT INTERESTS
RECORD<1>=LNAME
RECORD<2>=FNAME
RECORD<3>=STREET
RECORD<4>=CITY
RECORD<5>=STATE

RECORD<6>=ZIP
RECORD<7>=1989
RECORD<8>=INTERESTS
WRITE RECORD TO FILE, 1111
PRINT
EXECUTE 'LIST SUN.MEMBER LNAME WITH FNAME EQ ELLEN'
```

This is the program output:

```
ENTER YOUR FIRST NAME
?ELLEN
ENTER YOUR LAST NAME
?KRANZER
ENTER YOUR ADDRESS (PLEASE WAIT FOR PROMPTS)
STREET ADDRESS
?3 AMES STREET
ENTER CITY
?CAMBRIDGE
ENTER STATE
?MA
ENTER ZIP CODE
?02139
ENTER YOUR INTEREST
?SAILING

SUN.MEMBER  LAST NAME.
1111 KRANZER

1 records listed.
```

Syntax

WRITEBLK *expression* ON *file.variable*
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the WRITEBLK statement to write a block of data to a file opened for sequential processing. Each WRITEBLK statement writes the value of *expression* starting at the current position in the file. The current position is incremented to beyond the last byte written. WRITEBLK does not add a newline at the end of the data.

file.variable specifies a file opened for sequential processing.

Note: On Windows NT systems, if you use the WRITEBLK statement to write to a 1/4-inch cartridge tape (60 MB or 150 MB) that you opened with the OPENDEV statement, you must specify the block size as 512 bytes or a multiple of 512 bytes.

The value of *expression* is written to the file, and the THEN statements are executed. If no THEN statements are specified, program execution continues with the next statement. If the file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

If either *expression* or *file.variable* evaluates to the null value, the WRITEBLK statement fails, the program terminates, and a run-time error message appears.

Example

```
OPENSEQ 'FILE.E','RECORD4' TO FILE ELSE ABORT
WEOFSEQ FILE
DATA1='ONE'
DATA2='TWO'
*
WRITEBLK DATA1 ON FILE ELSE ABORT
WRITEBLK DATA2 ON FILE ELSE ABORT
* These two lines write two items to RECORD4 in FILE.E without
* inserting a newline between them.
WEOFSEQ FILE
SEEK FILE,0,0 ELSE STOP
READSEQ A FROM FILE THEN PRINT A
* This reads and prints the line just written to the file.
```

This is the program output:

```
ONETWO
```

Syntax

WRITELIST *dynamic.array* ON *listname*

Description

Use the WRITELIST statement to save a list as a record in the &SAVEDLISTS& file.

dynamic.array is an expression that evaluates to a string made up of elements separated by field marks. It is the list to be saved.

listname is an expression that evaluates to

record.ID

or

record.ID account.name

record.ID is the record ID of the select list created in the &SAVEDLISTS& file. If *listname* includes *account.name*, the &SAVEDLISTS& file of the specified account is used instead of the one in the local account. If *record.ID* exists, WRITELIST overwrites the contents of the record.

If either *dynamic.array* or *listname* evaluates to the null value, the WRITELIST statement fails, the program terminates, and a run-time error message appears.

WRITESEQ statement

Syntax

WRITESEQ *expression* {ON | TO} *file.variable* [ON ERROR *statements*]

Description

Use the WRITESEQ statement to write new lines to a file opened for sequential processing. UniVerse keeps a pointer to the current position in the file while it is open for sequential processing. The OPENSEQ statement sets this pointer to the first byte of the file, and it is advanced by the READSEQ, READBLK, WRITESEQ, and WRITEBLK statements.

WRITESEQ writes the value of *expression* followed by a newline to the file. The data is written at the current position in the file. The pointer is set to the position following the newline. If the pointer is not at the end of the file, WRITESEQ overwrites any existing data byte by byte (including the newline), starting from the current position.

file.variable specifies a file opened for sequential access.

The value of *expression* is written to the file as the next line, and the THEN statements are executed. If THEN statements are not specified, program execution continues with the next statement. If the specified file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

If *expression* or *file.variable* evaluates to the null value, the WRITESEQ statement fails, the program terminates, and a run-time error message appears.

After executing a WRITESEQ statement, you can use the STATUS function to determine the result of the operation:

0 The record was locked before the WRITESEQ operation.

–2 The record was unlocked before the WRITESEQ operation.

File Buffering

Normally UniVerse uses buffering for sequential input and output operations. If you use the NOBUF statement after an OPENSEQ statement, buffering is turned off and writes resulting from the WRITESEQ statement are performed right away.

You can also use the FLUSH statement after a WRITESEQ statement to cause all buffers to be written right away.

For more information about buffering, see the FLUSH and NOBUF statements.

The ON ERROR Clause

The ON ERROR clause is optional in the WRITESEQ statement. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the WRITESEQ statement is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

Example

```
DATA 'NEW ITEM 1', 'NEW ITEM 2'
OPENSEQ 'FILE.E', 'RECORD1' TO FILE ELSE ABORT
READSEQ A FROM FILE ELSE STOP
*
FOR I=1 TO 2
   INPUT B
   WRITESEQ B TO FILE THEN PRINT B ELSE STOP
NEXT
*

CLOSESEQ FILE
END
```

This is the program output:

```
?NEW ITEM 1
NEW ITEM 1
?NEW ITEM 2
NEW ITEM 2
```

WRITESEQF statement

Syntax

WRITESEQF *expression* {ON | TO} *file.variable* [ON ERROR *statements*]
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the WRITESEQF statement to write new lines to a file opened for sequential processing, and to ensure that data is physically written to disk (that is, not buffered) before the next statement in the program is executed. The sequential file must be open, and the end-of-file marker must be reached before you can write to the file. You can use the FILEINFO function to determine the number of the line about to be written.

Normally, when you write a record using the WRITESEQ statement, the record is moved to a buffer that is periodically written to disk. If a system failure occurs, you could lose all the updated records in the buffer. The WRITESEQF statement forces the buffer contents to be written to disk; the program does not execute the statement following the WRITESEQF statement until the buffer is successfully written to disk. A WRITESEQF statement following several WRITESEQ statements ensures that all buffered records are written to disk.

WRITESEQF is intended for logging applications and should not be used for general programming. It increases the disk I/O of your program and therefore degrades performance.

file.variable specifies a file opened for sequential access.

The value of *expression* is written to the file as the next line, and the THEN statements are executed. If THEN statements are not specified, program execution continues with the next statement. If the specified file cannot be accessed or does not exist, the ELSE statements are executed; any THEN statements are ignored.

If *expression* or *file.variable* evaluates to the null value, the WRITESEQF statement fails, the program terminates, and a run-time error message appears.

The ON ERROR Clause

The ON ERROR clause is optional in the WRITESEQF statement. Its syntax is the same as that of the ELSE clause. The ON ERROR clause lets you specify an alternative for program termination when a fatal error is encountered while the WRITESEQF statement is being processed.

If a fatal error occurs, and the ON ERROR clause was not specified, or was ignored (as in the case of an active transaction), the following occurs:

- An error message appears.
- Any uncommitted transactions begun within the current execution environment roll back.
- The current program terminates.
- Processing continues with the next statement of the previous execution environment, or the program returns to the UniVerse prompt.

A fatal error can occur if any of the following occur:

- A file is not open.
- *file.variable* is the null value.
- A distributed file contains a part file that cannot be accessed.

If the ON ERROR clause is used, the value returned by the STATUS function is the error number.

Values Returned by the FILEINFO Function

Key 14 (FINFO$CURRENTLINE) of the FILEINFO function can be used to determine the number of the line about to be written to the file.

Example

In the following example, the print statement following the WRITESEQF statement is not executed until the record is physically written to disk:

```
WRITESEQF ACCOUNT.REC TO ACCOUNTS.FILE
   THEN WRITTEN = TRUE
   ELSE STOP "ACCOUNTS.FILE FORCE WRITE ERROR"
PRINT "Record written to disk."
```

WRITET statement

Syntax

WRITET [UNIT (*mtu*)] *variable*
{THEN *statements* [ELSE *statements*] | ELSE *statements*}

Description

Use the WRITET statement to write a tape record on a magnetic tape. The value of *variable* becomes the next tape record. *variable* is an expression that evaluates to the text to be written to tape.

The UNIT clause specifies the number of the tape drive unit. Tape unit 0 is used if no unit is specified. If the UNIT clause is used, *mtu* is an expression that evaluates to a code made up of three decimal digits, as shown in the following table:

***mtu* Codes**

Code	Available Options
m (mode)	0 = No conversion 1 = EBCDIC conversion 2 = Invert high bit 3 = Invert high bit and EBCDIC conversion
t (tracks)	0 = 9 tracks. Only 9-track tapes are supported.
u (unit number)	0 through 7

The *mtu* expression is read from right to left. If *mtu* evaluates to a one-digit code, it represents the tape unit number. If *mtu* evaluates to a two-digit code, the rightmost digit represents the unit number and the digit to its left is the track number.

If either *mtu* or *variable* evaluates to the null value, the WRITET statement fails, the program terminates, and a run-time error message appears.

Each tape record is written completely before the next record is written. The program waits for the completion of data transfer to the tape before continuing.

Before a WRITET statement is executed, a tape drive unit must be attached (assigned) to the user. Use the ASSIGN command to assign a tape unit to a user. If no tape drive unit is attached or if the unit specification is incorrect, the ELSE statements are executed.

The largest record that the WRITET statement can write is system-dependent. If the actual record is larger, bytes beyond the system byte limit are not written.

Note: UniVerse BASIC does not generate tape labels for the tape file produced with the WRITET statement.

After a WRITET statement is executed, you can use the STATUS function to determine the result of the operation. The value returned by the STATUS function for the WRITET statement is hardware-dependent (that is, it varies according to the characteristics of the specific tape drive). Consult the documentation that accompanied your tape drive for information about interpreting the values returned by the STATUS function.

PIOPEN Flavor

If you have a program that specifies the syntax UNIT *ndmtu*, the *nd* elements are ignored by the compiler and no errors are reported.

Examples

The following example writes a record to tape drive 0:

```
RECORD=1s2s3s4
WRITET RECORD ELSE PRINT "COULD NOT WRITE TO TAPE"
```

The next example writes the numeric constant 50 to tape drive 2, a 9-track tape with no conversion:

```
WRITET UNIT(002) "50" ELSE PRINT "COULD NOT WRITE"
```

WRITEU statement

Use the WRITEU statement to maintain an update record lock while performing the WRITE statement. For details, see the WRITE statement.

WRITEV statement

Use the WRITEV statement to write on the contents of a specified field of a record of a UniVerse file. For details, see the WRITE statement.

WRITEVU statement

Use the WRITEVU statement to maintain an update record lock while writing on the contents of a specified field of a record of a UniVerse file. For details, see the WRITE statement.

Syntax

XLATE ([DICT] *filename*, *record.ID*, *field#*, *control.code*)

Description

Use the XLATE function to return the contents of a field or a record in a UniVerse file. XLATE opens the file, reads the record, and extracts the specified data.

filename is an expression that evaluates to the name of the remote file. If XLATE cannot open the file, a run-time error occurs, and XLATE returns an empty string.

record.ID is an expression that evaluates to the ID of the record to be accessed.

field# is an expression that evaluates to the number of the field from which the data is to be extracted. If *field#* is –1, the entire record is returned, except for the record ID.

control.code is an expression that evaluates to a code specifying what action to take if data is not found or is the null value. The possible control codes are:

X (default) Returns an empty string if the record does not exist or data cannot be found.

V Returns an empty string and produces an error message if the record does not exist or data cannot be found.

C Returns the value of *record.ID* if the record does not exist or data cannot be found.

N Returns the value of *record.ID* if the null value is found.

Note that the returned value is lowered. For example, value marks in the original field become subvalue marks in the returned value. For more information, see the LOWER function in Volume 1.

If *record.ID* is multivalued, the translation occurs for each record ID and the result is multivalued (value marks separate data translated from each record).

If *filename*, *record.ID*, or *field#* evaluates to the null value, the XLATE function fails, the program terminates, and a run-time error message appears. If *control.code* evaluates to the null value, null is ignored and X is used.

The XLATE function is the same as the TRANS function.

Example

```
X=XLATE("VOC","EX.BASIC",1,"X")
PRINT "X= ":X
*
FIRST=XLATE("SUN.MEMBER","6100",2,"X")

LAST=XLATE("SUN.MEMBER","6100",1,"X")
PRINT "NAME IS ":FIRST:" ":LAST
```

This is the program output:

```
X= F BASIC examples file
NAME IS BOB MASTERS
```

Syntax

XTD (*string*)

Description

Use the XTD function to convert a string of hexadecimal characters to an integer. If *string* evaluates to the null value, null is returned.

Example

```
Y = "0019"
Z = XTD (Y)
PRINT Z
```

This is the program output:

```
25
```

Index

Symbols

! statement 2
#INCLUDE statement 3
$CHAIN statement 4
$COPYRIGHT statement 5
$DEFINE statement 6
$EJECT statement 8
$ELSE statement 9
$ENDIF statement 10
$IFDEF statement 11
$IFNDEF statement 12
$INCLUDE statement 13
$INSERT statement 14
$OPTIONS statement 16
    default settings 22
    options 17–21
$PAGE statement 25
$UNDEFINE statement 26
* statement 27
< > operator 28, 166, 420
@ expressions in INPUT
        statements 258
@ function 29
@AUTHORIZATION variable 64
@NULL variable 61
@NULL.STR variable 61, 89, 90
@RECORD variable
    and ITYPE function 277
@SENTENCE variable 456
@STDFIL variable 349, 357
[ ] operator 49

A

ABORT statement 51
ABORTE statement 21, 51
ABORTM statement 21, 51
ABS function 52
ABSS function 53
ACOS function 54
ADDS function 55
ALPHA function 56
AND operator 57, 68
ANDS function 57
angle brackets 28, 166, 420
arrays
    assigning values to 306
    dimensioned 138
    passing to subroutines 81, 139, 492
ASCII function 58
ASIN function 59
ASSIGNED function 60
assignment operators 28, 61
assignment statements 61, 289
ATAN function 63
AUTHORIZATION statement 64
    and SELECT statements 448

B

BASIC programs
    conditional compiling 243
BEGIN CASE statement 83
BITAND function 68
BITNOT function 69
BITOR function 70
BITRESET function 71
BITSET function 72
BITTEST function 73
BITXOR function 74
blank spaces

removing 540, 541, 542, 543, 544, 545
brackets
angle 28, 166, 420
square 49
BREAK statement 75
BSCAN statement 77
B-tree files 77

C

CALL statement 80
and RETURN statement 424
and SUBROUTINE statement 492
CASE
option 17
statement 83
CATS function 86
CHAIN statement 87
CHANGE function 88
CHAR function 89
CHAR(128) 89, 90, 172
CHARS function 90
CHECKSUM function 91
CLEAR statement 92
CLEARDATA statement 93
CLEARFILE statement 94
CLEARPROMPTS statement 96, 247
CLEARSELECT statement 97
CLOSE statement 99
CLOSESEQ statement 101
COL1 function 103
COL2 function 104
comments 2, 27, 414
COMMIT statement 106
common
clearing 92
unnamed
saving variable values 87
COMMON statement 108
COMP.PRECISION option 17
COMPARE function 109
conditional compiling 243
configurable parameters
OPENCHK 237, 322, 350, 562
CONTINUE statement 204, 299
conversion codes
in format expressions 194
in ICONV function 239
in ICONVS function 241
in OCONV function 342
in OCONVS function 345
CONVERT
function 111
statement 112
COS function 113
COSH function 114
COUNT function 115
COUNT.OVLP option 17, 115, 117, 124, 250
COUNTS function 117
CREATE statement 119
and NOBUF statement 336
CRT statement 120
cursor positioning 29

D

DATA statement 121
and INPUT statements 258
DATE function 123
dates
internal system 123
DCOUNT function 124
DEBUG statement 125
debugger 125
DEFFUN statement 127, 209
DEL statement 129
DELETE
function 131
statement 133
DELETELIST statement 136
DELETEU statement 133, 137
DIM statement 138

DIMENSION statement 138
DISPLAY statement 141
distributed files
  status 482
DIV function 142
DIVS function 143
DOWNCASE function 144
DQUOTE function 145
DTX function 146
dynamic arrays
  passing to subroutines 81

E

EBCDIC function 147
ECHO statement 148
effective UID 64
  AUTHORIZATION statement 64
END CASE statement 83
END statement 149
END.WARN option 17, 149
ENTER statement 152
EOF(ARG.) function 153
EQS function 154
EQUATE statement 155
EREPLACE function 157
ERRMSG
  codes 158
  file 158, 374
    and STOPE statement 485
  statement 158
    and STOP statement 485
error messages 51, 158, 374, 485
EXCHANGE function 160
EXEC.EQ.PERF option 17, 163
EXECUTE statement 161
EXIT statement 164
EXP function 165
expressions
  @ 258
  format 191–194
external subroutines
  entering 152
EXTRA.DELIM option 17, 269, 272, 421
EXTRACT function 28, 166

F

FADD function 168
FDIV function 169
FFIX function 170
FFLT function 171
FIELD function 172
  and COL1 function 103
  and COL2 function 104
FIELDS function 174
file locks
  types 182
FILELOCK statement 182
files
  closing 99
  configuration information 177
  ERRMSG 158, 374
  locking 185
  sequential processing 101, 119, 190, 336, 355, 360, 390, 397, 445, 526, 570, 572
FILEUNLOCK statement 185
FIND statement 187
FINDSTR statement 188
FIX function 189
flavors 16
FLUSH statement 190
  after WRITESEQ statement 570
FMT function 191
FMTS function 196
FMUL function 197
FOOTING statement 199
FOR statement 203
FOR.INCR.BEF option 17, 204
format expressions 191–194
  in INPUT statements 259
FORMAT.OCONV option 17

FORMLIST statement 207
FSELECT option 17
FSUB function 208
FUNCTION statement 209
functions
    user-written 426

G

GES function 211
GET statement 212
GET(ARG.) statement 218
GETLIST statement 220
GETREM function 221
GETX statement 212, 217
GOSUB statement 222
    and RETURN statement 424
    with ON statement 346
GOTO statement 224
    with ON statement 347
GROUP function 225
GROUPSTORE statement 227
GTS function 229

H

HEADER.BRK option 17
HEADER.DATE option 18, 202, 233
HEADER.EJECT option 18, 231
HEADING statement 230
HEADINGE statement 21, 231
HEADINGN statement 21, 231
host name
    and SYSTEM function 500
HUSH statement 235

I

ICHECK function 237
ICONV function 239
ICONVS function 241
IF statement 243
IFS function 245
ILPROMPT function 246
IN2.SUBSTR option 18
INCLUDE statement 249
INDEX function 250
INDEXS function 252
INDICES function 253
    in transactions 255
INFO.ABORT option 18
INFO.CONVERT option 18
INFO.ENTER option 18
INFO.INCLUDE option 18
INFO.LOCATE option 18, 293
INFO.MARKS option 18
INFO.MOD option 18
in-line prompts
    clearing 96
INMAT function 256
    after MATPARSE statement 314
    after MATREAD statement 317
    after OPEN statement 350
INPUT @ statement 258
INPUT statements 258
    and DATA statement 121
INPUT.ELSE option 19, 260
INPUTAT option 18
INPUTCLEAR statement 262
INPUTDISP statement 263
INPUTERR statement 264
INPUTIF statement 258
INPUTNULL statement 266
INPUTTRAP statement 267
INS statement 268
INSERT function 271
    and LOCATE statement 292
INT function 274
INT.PRECISION option 19
internal system date 123
ISNULL function 275
    with CASE statement 83
    with IF statement 243
ISNULLS function 276

ITYPE function 277

K

KEEP.COMMON keyword 87
KEYEDIT statement 279
KEYEXIT statement 281
KEYIN function 282

L

LEFT function 285
LEN function 286
LENS function 287
LES function 288
LET statement 289
LN function 290
local variables 87
LOCATE statement 291
LOCATE.R83 option 19
LOCK statement 296
locks
    and MATREADL statement 317
    and MATREADU statement 317
    and READL statement 386
    and READU statement 324, 386, 564
    and READVL statement 386
    file 185
    process 557
    releasing 324, 388, 410, 411, 557, 564
    semaphore 296
    types 182
LOOP statement 298
loops
    FOR...NEXT 203, 335
    LOOP...REPEAT 298
LOWER function 301
LTS function 303

M

MAT statement 306
MATBUILD statement 308
MATCH operator 309
MATCHFIELD function 311
MATPARSE statement 313
    and INMAT function 256
MATREAD statement 316
    and INMAT function 256
MATREADL statement 317
    and INMAT function 256
MATREADU statement
    and INMAT function 256
MATWRITE statement 322
MATWRITEU statement 324
MAXIMUM function 326
messages
    error 51, 158, 374, 485
MINIMUM function 327
MOD function 328
MODS function 329
MULS function 330

N

NAP statement 331
NEG function 332
NEGS function 333
NES function 334
nested transactions
    committing 106
NEXT statement 203, 335
NO.CASE option 19
NO.RESELECT option 19
NOBUF statement 336
    used with TTYSET 555
NOT function 337
NOTS function 338
NULL statement 339
NUM function 340
NUMS function 341

O

OCONV function 342
OCONVS function 345
ON statement 346
ONGO.RANGE option 19, 347
OPEN statement 349
    and INMAT function 256
OPENCHECK statement 237, 353
OPENCHK parameter 237, 322, 350, 562
OPENDEV statement 355
OPENPATH statement 357
OPENSEQ statement 360
    and CREATE statement 119
    and NOBUF statement 336
operators
    assignment 28, 61
OR operator 70, 365
ORS function 365

P

PAGE statement 366
part files
    status 482
part numbers
    status 482
pattern matching 309
    table of codes 309
PCLOSE.ALL option 19
PERF.EQ.EXEC option 19, 368
PERFORM statement 367
PIOPEN.INCLUDE option 19
PIOPEN.SELIDX option 19, 452
pointer (REMOVE) 221, 429, 461
PRECISION statement 369
PRINT statement 370
    and INPUT statements 259
    and TABSTOP statement 502
PRINTER CLOSE statement 372
PRINTER statement 372
PRINTERR statement 374
process locks 557
PROCREAD statement 376
PROCWRITE statement 377
PROGRAM statement 378
PROMPT statement 379
    and INPUT statements 259
prompts, *see* in-line prompts
PWR function 380

Q

QUOTE function 381

R

RADIANS option 19
RAID (debugger) 125
    table of commands 125
RAISE function 382
RANDOMIZE statement 384
    and RND function 433
RAW.OUTPUT option 20
READ statement 385
READ.RETAIN option 20, 388
READBLK statement 390
    and TIMEOUT statement 526
READL statement 385
READLIST statement 393
READNEXT statement 395
    and READLIST statement 393
    and SELECT statement 448
READSEQ statement 397
    and TIMEOUT statement 526
READT statement 399
READU statement 385
READV statement 385
READVL statement 385
READVU statement 385
REAL function 405
REAL.SUBSTR option 20
RECORDLOCKED function 406
RECORDLOCKL statement 408
RECORELOCKU statement 408

RELEASE statement 411
REM
  function 413
  statement 414
REMOVE
  function 415
  pointer 221, 429, 461
  statement 417
removing
  spaces 540, 541, 542, 543, 544, 545
  tabs 540, 541, 542, 543, 544, 545
REPEAT statement 298
REPLACE function 28, 166, 420
RETURN (*value*) statement 209
  and DEFFUN statement 127
RETURN statement 81, 222, 424
RETURN TO statement 222
REUSE function 427
REVREMOVE statement 429
REWIND statement 431
RIGHT function 432
RND function 433
  and RANDOMIZE statement 384
RNEXT.EXPL option 20, 395
ROLLBACK statement 434
RPC.CALL function 436
RPC.CONNECT function 438
RPC.DISCONNECT function 440

S

SADD function 441
SCMP function 442
SDIV function 443
secondary indices
  and BASIC INDICES function 253
SEEK statement 444
SEEK(ARG.) statement 446
select lists
  clearing 97
  variables 97, 451, 472
SELECT statement 448
SELECTE statement 451
SELECTINDEX statement 452
SELECTINFO function 454
SELECTN statement 21, 450
SELECTV statement 21, 449
semaphore locks 296
SEND statement 455
SENTENCE function 456
SEQ function 457
SEQ.255 option 20, 457
SEQS function 458
sequential processing 101, 119, 190, 336, 355, 360, 390, 397, 445, 526, 570, 572
SET TRANSACTION ISOLATION LEVEL statement 459
SETREM statement 461
SIN function 462
SINH function 463
SLEEP statement 464
SMUL function 465
SOUNDEX
  function 466
SPACE function 467
spaces
  removing 540, 541, 542, 543, 544, 545
SPACES function 468
SPLICE function 469
SQRT function 470
square brackets 49
SQUOTE function 471
SSELECT statement 472
SSELECTN statement 474
SSELECTV statement 472
SSUB function 475
STATIC.DIM option 20, 139
STATUS
  function 476
    after BSCAN statement 78, 476
    after DELETE statement 476
    after FILELOCK statement 476

after FMT function 476
after GET statement 477
after GETX statement 477
after HUSH statement 235
after ICONV function 239, 477
after ICONVS function 241
after INPUT @ statement 477
after KEYEXIT statement 281
after KEYTRAP statement 283
after MATWRITE statement 477
after OCONV function 342, 477
after OCONVS function 345
after OPEN statement 358
after OPEN statements 477
after READ statement 478
after READBLK statement 478
after READL statement 479
after READSEQ statement 397, 479
after READT statement 400, 479
after READU statement 479
after READVL statement 479
after READVU statement 479
after RECORDLOCKED statement 407
after REWIND statement 431, 479
after RPC.CALL function 436
after RPC.CALL statement 479
after RPC.CONNECT function 438
after RPC.CONNECT statement 479
after RPC.DISCONNECT function 440
after RPC.DISCONNECT statement 479
after SELECTINDEX statement 452
after WEOF statement 479
after WRITE statements 477
after WRITESEQ statement 570
after WRITET statement 479, 575
statement 481
table of values 481
STOP statement 485
STOP.MSG option 20, 51, 485
STOPE statement 21, 485
STOPM statement 21, 485
STORAGE statement 487
STR function 488
STRS function 489
SUBR function 490
SUBROUTINE statement 80, 492
and SUBR function 490
subroutines
calling 80
entering external 152
passing arrays to 81, 139
returning from 424
returning values from 490
SUBS function 493
substrings
extracting 49
SUBSTRINGS function 494
SUM function 495
SUMMATION function 497
SUPP.DATA.ECHO option 20, 260
syntax notation xv
system date 123
SYSTEM function 498
host name 500
table of values 498

T

tabs
removing 540, 541, 542, 543, 544, 545
TABSTOP statement 502
and PRINT statement 370
and TPRINT statement 531
TAN function 503
TANH function 504

TERMINFO function 505
table of EQUATES 505–522
tilde (~) 309
TIME function 524
TIME.MILLISECOND option 20
TIMEDATE function 525
TIMEOUT statement 526
TPARM function 528
TPRINT statement 531
TRANS function 533
TRANSACTION ABORT statement 536
TRANSACTION COMMIT statement 538
TRANSACTION START statement 539
transaction statements 535
transactions
and RELEASE statements 411
and UV/Net 562
TRIM function 540
TRIMB function 541
TRIMBS function 542
TRIMF function 543
TRIMFS function 544
TRIMS function 545
TTYCTL statement 546
TTYGET statement 548
table of values 549
TTYSET statement 554

U

UID
effective 64
ULT.FORMAT option 20
UNASSIGNED function 556
UNLOCK statement 557
and LOCK statement 296
unnamed common
saving variable values 87
UNTIL statement 203, 298
UPCASE function 558
USE.ERRMSG option 20, 375
user ID 64
user-written functions 426
UV/Net
and transactions 562
AUTHORIZATION statement 64
RPC.CONNECT function 438
SYSTEM function 500
TIMEOUT statement 526
WRITE statement 562
writing to remote files 562

V

VAR.SELECT option 21, 97, 220, 393, 394, 449, 450, 473, 474
variables
assigning 61, 289
common 108
in user-written functions 209
local 87
passing to subroutines 139, 492
saving in unnamed common 87
select list 97, 451, 472
unnamed common 87
VEC.MATH option 21

W

WEOF statement 559
WEOFSEQ statement 559, 560
WHILE statement 203, 298
WIDE.IF option 21
WRITE statement 562
WRITEBLK statement 567
WRITELIST statement 569
WRITESEQ statement 570, 572
WRITESEQF statement 572
WRITET statement 574
WRITEU statement 562
WRITEV statement 562

WRITEVU statement 562

X

XLATE function 579
XTD function 581

To help us provide you with the best documentation possible, please make your comments and suggestions concerning this manual on this form and fax it to us at **508-366-3669, attention Technical Publications Manager**. All comments and suggestions become the property of VMARK SOFTWARE, INC. We greatly appreciate your comments.

Comments

Name: _____ Date: _____

Position: _____ Dept: _____

Organization: _____ Phone: _____

Address: _____

_____

Name of Manual: **UniVerse BASIC Reference Volume 2 Volume 1**

Part Number: **70-9002A-931**